Surrogate-Assisted Multi-objective Optimization for Compiler Optimization Sequence Selection

Guojun Gao[1,2], Lei Qiao[3(✉)], Dong Liu[1,2], Shifei Chen[1,2], and He Jiang[1,2(✉)]

[1] School of Software, Dalian University of Technology, Dalian, China
{ggj_gao,chenshifei}@mail.dlut.edu.cn, {dongliu,jianghe}@dlut.edu.cn
[2] Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, Dalian, China
[3] Beijing Institute of Control Engineering, Beijing, China
fly2moon@aliyun.com

Abstract. Compiler developers typically design various optimization options to produce optimized programs. Generally, it is a challenging task to identify a reasonable set of optimization options (i.e., compiler optimization sequence) in modern compilers. Optimization objectives, in addition to the target architecture and source code of the program, influence the selection of optimization sequences. Current applications are often required to optimize two or more conflicting objectives simultaneously, such as execution time and code size. Existing approaches employ evolutionary algorithms to find appropriate optimization sequences to trade off the above two objectives. However, since program compilation and execution are time-consuming, and the two objectives are inherently conflicting, applying evolutionary algorithms faces the diverse objectives influence and computationally expensive problem. In this study, we present a surrogate-assisted multi-objective optimization approach. To speed up the convergence, it employs a fast global search based on non-dominated sorting. The approach then uses two surrogate models for each objective to generate approximate fitness evaluations rather than using actual expensive evaluations. Extensive experiments on the benchmark suite cBench show that our approach outperforms the baseline NSGA-II on hypervolume by an average of 11.7%. Furthermore, experiments verify that the surrogate model contributes to solving the computationally expensive problem and taking fewer actual fitness evaluations.

Keywords: Multi-objective · Compiler optimization sequence selection · Surrogate model

1 Introduction

Today, the compiler is one of the most important foundations of the complex software infrastructure, and it has been used to generate optimized executable binaries for several decades [4,14]. Modern compilers provide numerous compiler

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
G. Rudolph et al. (Eds.): PPSN 2022, LNCS 13399, pp. 382–395, 2022.
https://doi.org/10.1007/978-3-031-14721-0_27

optimization options to satisfy a wide range of complex optimization requirements (e.g., code size and execution time). GCC, for example, offers hundreds of optimization options. As a result, it is impractical to select the best compiler optimization sequence from massive optimization options to optimize programs by hand. Despite the fact that compilers provide some predefined standard optimization levels (-O1,-O2,-O3,-Os, etc.) with a fixed optimization sequence, they fail to achieve the best performance on every program [3,4,9].

Besides, when selecting an appropriate compiler optimization sequence, the selection is indeed influenced not only by the program source code and the target architecture but also by the optimization objectives [8]. For the compiled object code, however, the code size and execution time are two conflicting objectives. The program will be expanded if you pursue the program's execution time. On the contrary, pursuing smaller executable code frequently results in slower program execution speed [21].

In previous studies, multi-objective optimization algorithms were used to select optimization sequences to make a trade-off between code size and execution time. Lokuciejewski et al. [16,17] applied SPEA2, NSGA-II, and IBEA to select optimization sequences that reduced execution time and code size. Experiments show that the performance of these algorithms is significantly improved compared with standard optimization levels. Chebolu and Wankar [8] used a novel genetic algorithm based on weighted value functions to obtain a faster execution time than the binary code generated by "-Ofast", while the code size does not exceed "-OS". Ansel et al. [2] introduced "OpenTuner", an open-source framework for constructing multi-objective optimized compiler optimization sequences. The framework supports the user-defined setting of multiple search methods to search for the appropriate optimization sequence.

However, the key issue in applying the multi-objective optimization algorithm in selecting compiler optimization sequences is its efficiency. On the one hand, multi-objective optimization needs to search the whole objective space, and the conflicting objectives may influence the convergence rate to some extent [15]. On the other hand, because compiling and executing a program is time-consuming, the computational cost of fitness function evaluations is high, and the number of evaluations required to obtain Pareto-optimal solutions is restricted. This brings up the computationally expensive problem.

To address the above two challenges, we propose a novel surrogate-assisted multi-objective optimization approach that efficiently selects promising optimization sequences. The approach contains two key components: a fast global search and surrogate models. Firstly, a fast global search based on non-dominated sorting is adopted to deliberate the overall performance on different objectives. Secondly, surrogate models are employed to approximate the expensive fitness functions to avoid a lot of actual compilation and execution. In the iterative process, two surrogate models for execution time and code size are used to determine which individuals can be incorporated into the non-dominated solution set. These individuals are then compiled and executed to obtain fitness values. Our goal is to develop an efficient approach for the computationally expensive multi-objective problem with over one hundred variables. Here, we use random forest surrogate models to approximate the expensive compilation.

Furthermore, we study our approach experimentally on the compiler GCC and the benchmark suite cBench [11]. Experimental results demonstrate that our approach performs averagely better than NSGA-II by 11.7% on hypervolume. Besides, the impact of the parameters: crossover rate and mutation rate, is investigated. It shows that configuring the two parameters has no noticeable effect on our approach. Additionally, a comparison between the convergence rate of our approach and a variant without surrogate models indicates that the surrogate models contribute to the efficiency of compiler optimization sequence selection. The contributions of this paper are summarized as follows:

- A novel approach is proposed for efficiently selecting optimization sequences to reduce program code size and speed up execution.
- Surrogate models and a fast global based on non-dominated sorting are developed to overcome the challenges of the computationally expensive problem and the influence of diverse objectives.
- Experimental studies conducted on the compiler GCC and the benchmark suite cBench show that our approach not only identifies better compiler optimization sequences, but also involves fewer expensive fitness evaluations.

The remainder of this paper is organized as follows. Section 2 describes the background of the compiler optimization sequence selection. Section 3 provides a detailed description of the proposed approach. Studies comparing the proposed approach with NSGA-II and the experimental results are presented in Sect. 4. Finally, Sect. 5 concludes the paper with a summary and some ideas for future work.

2 Background

In this section, we introduce the background knowledge of compiler optimization sequence selection as well as the multi-objective optimization for the compiler optimization sequence selection.

2.1 Compiler Optimization Sequence Selection

The compiler provides plentiful optimization options to satisfy different performance requirements. A set of compiler optimization options forms an optimization sequence. We define the problem as **compiler optimization sequence selection** if we disregard the order of the compiler optimization options and instead focus on whether an optimization option is applied. Many previous studies [1,5] have shown that the interactions among optimization options are so complicated that they have a significant impact on program performance.

The problem of compiler optimization sequence selection can be formalized as follows:

Let a Boolean vector $seq = \{o_1, o_2, o_3, ..., o_n\}$ be a compiler optimization sequence, and n is the number of optimization options under analysis. The ith

element o_i in seq represents the ith optimization option used. Besides, the value of an optimization option o_i is $o_i = 1$ or $o_i = 0$, indicating whether the optimization option is turned on or off.

Given a program P that is being optimized, one or more of the effective optimization sequences $\overline{seq}$ will be found and provided to the compiler to generate smaller or faster machine object code.

Furthermore, the search space $S = \{0, 1\}^n$ of the problem has an exponential space. For instance, if we only analyze $n = 10$ optimization options and select the right optimization sequence, we need to explore a total state space of $2^n = 1024$. In practice, there are far more than ten, even hundreds of optimization options available when using the compiler to optimize the program. Thus, developers face a significant challenge in manually selecting an appropriate optimization sequence. Automatic methods are necessary to be introduced.

2.2 Multi-objective Optimization for Compiler Optimization Sequence Selection

This study focuses on the optimization of execution time and code size of the machine object code. The program's execution time should be as short as possible during the process of compiler optimization sequence selection using multi-objective optimization. Additionally, to save storage space, the code size should be as small as possible. For easy understanding and unified representation, we design two fitness functions to calculate execution speedup and code size reduction, respectively (as shown in Eqs. 2 and 3 in Sect. 3). In summary, we set the maximum execution speedup and the maximum code size reduction as two optimization objectives in our study.

The multi-objective optimization problem of compiler optimization sequence selection is formulated in Eq. 1 as follows:

$$\max_{seq \in S} (Fitness_s(seq), Fitness_t(seq)) \tag{1}$$

where $Fitness_s(seq)$ is the fitness functions for code size and $Fitness_t(seq)$ is the fitness functions for execution time. The compiler optimization sequence is denoted by seq, and the search space is denoted by the set S. Finding a feasible compiler optimization sequence that maximizes code size reduction and execution speedup at the same time is unthinkable. As a result, we seek to investigate and identify Pareto optimal solutions that cannot be improved in any objective without degrading others.

3 The Proposed Approach

This paper proposes a surrogate-assisted multi-objective optimization algorithm for optimization sequence selection to enhance the performance in terms of execution time and code size. In this section, we will first present the solution representation and the fitness function in our approach. Following that, our approach is described in detail. Finally, we introduce the surrogate model that we used in our approach.

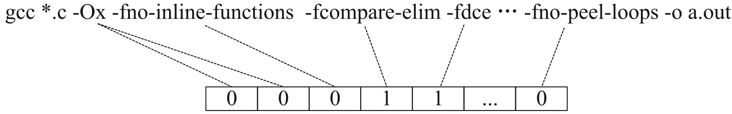

Fig. 1. The solution representation

3.1 Representation

For the current study of selecting the best optimization sequences, a candidate solution represents an optimization sequence that is composed of optimization options from the four standard optimization levels (-O1,-O2,-O3, and -Os). The solution representation in this paper is shown in Fig. 1.

Here, we use a vector to represent a solution, with the first two dimensions representing the encoding of four standard optimization levels and the remaining dimensions representing compiler options. For example, we set the first two dimensions to '00', '01', '10', and '11' to represent -O1, -O2, -O3, and -Os, respectively. The rest dimensions' values are Boolean values that are represented as genes on a chromosome. We set the value to 1 or 0, where 1 indicates that we select the optimization option to optimize the program and 0 indicates that the optimization option is disabled. As illustrated in Fig. 1, the compiler is then informed of the optimization selection information via -f⟨optimization option⟩ and -fno-⟨optimization option⟩, which indicate the selection or non-selection of the optimization option.

Besides, a solution has a fixed size, which includes two label bits (the first two dimensions) and all optimization options used in this study. Because the compiler GCC has its own logic when invoking the optimization option, the order of the optimization options that we provided has no effect on the results. We keep the same order of the optimization options that the compiler is given.

3.2 Fitness Function

The fitness function is intended to assess the quality of a candidate solution. In our study, we look at two objectives: execution time and code size. To measure the code size reduction and execution time speedup of a candidate optimization sequence seq, two fitness functions $Fitness_s(seq)$ and $Fitness_t(seq)$ are defined. As shown in Eqs. 2 and 3, the two fitness functions use the default -O0 and -O3 as bases, respectively. The optimization level -O0 does not optimize the program at all, while -O3 provides the most aggressive optimization on execution time.

$$Fitness_s(seq) = code_size(\text{-O0}) - code_size(seq) \tag{2}$$

where $code_size(seq)$ represents the code size of the executable file when the optimization sequence seq is applied to the program. We apply the 'size' command to the generated executable file, which yields the code size corresponding to the optimization sequence.

$$Fitness_t(seq) = execution_time(\text{-O3})/execution_time(seq) \tag{3}$$

where $execution_time(seq)$ represents the execution time of the executable file. Similarly, we use the 'time' command to get the execution time while the executable file is running.

3.3 Surrogate-Assisted Multi-objective Optimization Algorithm

We develop a novel surrogate-assisted multi-objective optimization algorithm to address the challenges of diverse objectives influence and the computationally expensive problem mentioned in Sect. 1. The pseudo-code of the proposed approach is presented in Algorithm 1.

First of all, the approach generates some parameters and functions, such as population size p_{num}, crossover rate p_{α}, mutation rate p_{β}, optimization sequence seq, program P, and the fitness functions for execution time $Fitness_t(seq)$ and code size $Fitness_s(seq)$. A set of solutions in *population* is randomly initialized before the iteration begins (line 2). These solutions are then evaluated using the actual expensive fitness functions (i.e., $Fitness_t(seq)$ and $Fitness_s(seq)$). The non-dominated solutions are archived in *Archive* (lines 3–5). Meanwhile, two surrogate models for execution time and code size are constructed in line 6 to predict the fitness values of optimization sequences.

In the main loop (lines 7–22), we apply crossover, mutation, and selection to generate a new population in the evolutionary process. Concretely, the crossover operator and mutation operator are utilized to produce offspring (lines 8–9). Here, we use the traditional single-point crossover [10] to generate child solutions and bit flip mutation to mutate the solution. Then, using two surrogate models, $Surrogate_t()$ and $Surrogate_s()$, we evaluate all solutions in the offspring and obtain their approximate execution time and code size (line 10).

Afterward, the new population is reproduced by combining the parent and offspring populations using $nondominated_sorting(population)$ and $selection$ $(population, p_{num})$ (lines 11–13). Before applying the selection mechanism, a fast non-dominated sorting procedure is used in *population* to divide the population into non-dominated fronts $F = \{F1, F2, ..., Fn\}$. The non-dominated solutions in *population* belong to the front $F1$. The solutions in $F1$ are then discounted temporarily, and the non-dominated solutions in the remaining population form the next front $F2$. Repeating the above procedure until all solutions in *population* are assigned to a front. During the selection process, solutions are chosen from the front $F1$ to Fn, which can help maintain the elitism while also generating good solutions. Furthermore, for solutions along the same front, crowding distance is used to select suitable solutions until the number of *population* reaches p_{num}.

Following that, all solutions in the new population that are offspring are reevaluated using $Fitness_t(seq)$ and $Fitness_s(seq)$ to get their actual fitness values (lines 14–18). In lines 19–20, the new population is sorted again, and the non-dominated solutions are saved in *Archive*. Next, two surrogate models, $Surrogate_t()$ and $Surrogate_s()$ are updated until they achieve a high level of accuracy. When the program reaches the stopping criterion, the evolutionary process is terminated.

Finally, the achieved solutions in *Archive* are regarded as Pareto optimal solutions.

Algorithm 1: Surrogate-assisted Multi-objective optimization algorithm

```
Input: population size p_num, crossover rate p_α, mutation rate p_β, optimization sequence
       seq, program P, the fitness function of the objective execution time Fitness_t(seq),
       the fitness function of the objective code size Fitness_s(seq)
Output: pareto optimal solutions Archive
1  begin
2      population ⟵ Initialize(p_num)
3      Evaluate population using Fitness_t(seq) and Fitness_s(seq)
4      population ⟵ nondominated_sorting(population)
5      Archive ⟵ the_non-dominated_solutions in population
6      Construct two surrogate models for each objective, Surrogate_t() and Surrogate_s()
       // Generating the next population in the evolutionary process
7      while Stopping criterion is not met do
8          offspring ⟵ crossover(population, p_α)
9          offspring ⟵ mutation(offspring, p_β)
10         Evaluate offspring using two surrogate models Surrogate_t() and Surrogate_s()
11         population ⟵ population ∪ offspring
12         population ⟵ no − ndominated_sorting(population)
13         population ⟵ selection(population, p_num)
14         for seq_i ∈ population do
15             if seq_i in offspring then
16                 Evaluate seq_i using Fitness_t(seq) and Fitness_s(seq)
17             end
18         end
19         population ⟵ nondominated_sorting(population)
20         Archive ⟵ the non-dominated solutions in population
21         Update two surrogate models Surrogate_t() and Surrogate_s()
22     end
23     return pareto optimal solutions Archive
24 end
```

3.4 Surrogate Model

Random Forest. The random forest [6] is an ensemble learning algorithm based on the bootstrap sampling technique that consists of several decision trees to solve classification or regression tasks. It is one of the most commonly used methods in numerous studies as the surrogate model [7,13,19]. For classification or regression tasks, the random forest constructs a multitude of decision trees and uses the result of most trees selected or the mean value of all trees returned to make the prediction. Therefore, the random forest outperforms any individual tree in terms of performance.

In this study, we also utilize the random forest as a surrogate model. Because we are concentrating on the compiler optimization options, and their values are Boolean values. Besides, the current study has demonstrated that the random forest as a surrogate model is very suitable for approximating such problems with discrete decision variables [13]. Additionally, the random forest has advantages in dealing with high-dimensional problems and can avoid over-fitting problems [12, 20]. Since over one hundred optimization options are considered in optimization sequence selection, we use the random forest in our work to approximate two objectives: the execution time and code size.

Surrogate Model Managing. As shown in Algorithm 1, two random forest models, $Surrogate_t()$ and $Surrogate_s()$, are constructed to approximate the execution time and code size rather than using expensive fitness evaluations. In the iterative procedure, we evaluate the solutions in offspring using two random forest models to obtain the predicted values of the two objectives. It should be noted that because the fitness values of all offspring solutions are approximated using surrogate models, the approximation error and prediction accuracy of surrogate models should be considered. We estimate two random forest models using the root mean square error (RMSE) and record $RMSE = \{R_1, R_2\}$. It is the square root of the difference between the predicted and actual value. The formula for calculation is shown below:

$$R_i = \sqrt{\frac{1}{N}\sum_{j=1}^{N}(\widehat{y_i^j} - y_i^j)^2} \quad (4)$$

where R_i is the root mean square error of the ith objective ($i = 1$ or 2), y_i^j represents the actual value of the ith objective of the jth solution, $\widehat{y_i^j}$ represents the predicted value of the ith objective of the jth solution obtained by the surrogate model, and N is the number of solutions in the current population. The lower the RSME, the better the predictive accuracy of the surrogate model.

Based on the mechanism of new population production, some solutions are introduced into the newly reproduced population by merging the parent population and the offspring, and then their actual fitness values are obtained using the actual compilation. These new reevaluated solutions with actual fitness values are employed to update two surrogate models until the RMSE threshold is satisfied. Hench, we train the surrogate models using the solutions in the initial population and these reevaluated solutions using fitness functions during iteration.

4 Experimental Results

4.1 Experimental Setup

In this section, we empirically evaluate the performance of our proposed approach and its ability to select promising compiler optimization sequences. Here, we investigate and answer the following Research Questions (RQs):

RQ1: How does our approach stack up against the baseline?
RQ2: How do parameters impact the outcome of our approach?
RQ3: How do surrogate models help our approach be more efficient?

For the above three RQs, RQ1 investigates the effectiveness of our approach and the quality of the selected compiler optimization sequences. RQ2 intends to analyze the impact of the choice of the crossover and mutation rate to set competitive parameters. RQ3 seeks to determine whether the surrogate models improve the efficiency of our approach.

Table 1. The programs in cBench benchmark suite

No.	Program	No.	Program	No.	Program
1	automotive_bitcount	12	consumer_tiff2bw	23	security_blowfish_e
2	automotive_qsort1	13	consumer_tiff2rgba	24	security_pgp_d
3	automotive_susan_c	14	consumer_tiffdither	25	security_pgp_e
4	automotive_susan_e	15	consumer_tiffmedian	26	security_rijndael_d
5	automotive_susan_s	16	network_dijkstra	27	security_rijndael_e
6	bzip2d	17	network_patricia	28	security_sha
7	bzip2e	18	office_ispell	29	telecom_adpcm_c
8	consumer_jpeg_c	19	office_ghostscript	30	telecom_adpcm_d
9	consumer_jpeg_d	20	office_rsynth	31	telecom_CRC32
10	consumer_lame	21	office_stringsearch1	32	telecom_gsm
11	consumer_mad	22	security_blowfish_d		

In this study, we conduct experiments on the benchmark suite cBench with the compiler GCC 9.4.0. cBench, which is commonly used in compiler auto-tuning involving various programs such as embedding functions and desktop programs. The 32 programs in cBench are listed in Table 1. Similarly, the employment of GCC is due to its popularity and provision of plenty of optimization options. The complete list of 107 optimization options is available at the GCC official website[1]. The experiments are then run 15 times with different random seeds to reduce the impact of their stochastic nature and produce reasonable results. In addition, all experiments are carried out on a machine equipped with an Intel Core i9 2.8 GHz CPU and 32 GB of memory running Ubuntu 20.04.

4.2 Experimental Results

Performance Metric. We use the popular hypervolume (HV) [22] to assess the quality of the Pareto front in the comparisons between our approach and the baseline. This metric measures the volume of the objective space between the Pareto front and the reference point. Before computing HV, we use the min-max normalization to normalize the objective values and choose the point [1,1] as the reference point to maintain the accuracy of HV. Due to the aim of maximizing both objectives, a lower HV value indicates higher quality.

Investigation of RQ1. To investigate whether our approach can effectively generate suitable optimization sequences in RQ1. We adopt the HV values to compare the Pareto set explored by our approach with the baseline NSGA-II. Besides, the Wilcoxon rank sum test with a significance level of 0.05 is used to assess the statistical significance of the difference between the two algorithms.

[1] https://gcc.gnu.org/onlinedocs/gcc-9.4.0/gcc/Optimize-Options.html#Optimize-Options.

Table 2. The statistical results of our approach and NSGA-II on cBench

No.	HV(NSGA-II)	HV(Our)	p-value	No.	HV(NSGA-II)	HV(Our)	p-value
1	0.5817	0.5148	1.16E−02	17	0.9092	0.8214	1.69E−05
2	0.4578	0.3994	4.89E−01	18	0.6261	0.5385	1.12E−04
3	0.1246	0.1075	3.48E−03	19	0.5737	0.4764	3.97E−04
4	0.2266	0.2042	1.58E−01	20	0.7986	0.7384	5.12E−05
5	0.2129	0.1898	2.18E−03	21	0.0341	0.0300	3.55E−04
6	0.5088	0.4699	7.48E−05	22	0.4466	0.3877	6.28E−03
7	0.4928	0.4413	4.57E−02	23	0.4358	0.3868	7.91E−06
8	0.5613	0.4736	1.91E−04	24	0.6511	0.5897	4.63E−06
9	0.3729	0.3384	7.36E−03	25	0.5632	0.5224	1.14E−04
10	0.6700	0.5732	7.16E−05	26	0.8474	0.7556	1.77E−01
11	0.5790	0.5301	3.21E−01	27	0.9125	0.8421	8.65E−06
12	0.8182	0.7040	6.66E−05	28	0.2798	0.2408	3.61E−05
13	0.7371	0.6694	7.85E−05	29	0.4991	0.4476	8.91E−06
14	0.6443	0.5462	2.01E−05	30	0.5113	0.4335	3.68E−05
15	0.5396	0.4827	2.50E−03	31	0.6537	0.5449	1.34E−05
16	0.0654	0.0554	3.17E−03	32	0.7065	0.5919	2.19E−05
Average					0.5325	0.4702	

In our experiments, the population size p_{num} is set to 100. Then, the crossover rate and mutation rate are set to 0.9 and 0.01, respectively, as the recommended parameter settings in RQ2 (the impact of crossover and mutation rate will be investigated later). Table 2 shows the HV values and p-values obtained using our approach and NSGA-II. The first and fifth columns represent the numerical order of the programs. The second and sixth columns are the HV values of NSGA-II. The HV values of our approach are then listed in columns three and seven. The p-values are shown in columns four and eight. Finally, the average HV values of the two algorithms are provided at the bottom of the table.

It can be seen from the table, our approach can explore a better Pareto set and achieve better performance in the majority of programs. On average, our approach achieves 0.4702, while NSGA-II achieves 0.5325. When compared to NSGA-II, our approach improves by 11.7% and performs reasonably well on HV values. There are two major reasons for this result. One is the utilization of random forest, as a surrogate model, which is appropriate for high-dimensional discrete problems and reduces the actual solution evaluations. Another is that the surrogate model management mechanism adds potential solutions to the training set to update the surrogate model, thus improving the search procedure and quality. Besides, as shown in Table 2, the p-values on 28 programs are less than 0.05, implying that there is a significant difference between our approach and NSGA-II for the majority of programs (28/32). In summary, these results reveal that our approach outperforms NSGA-II in terms of finding optimization sequences.

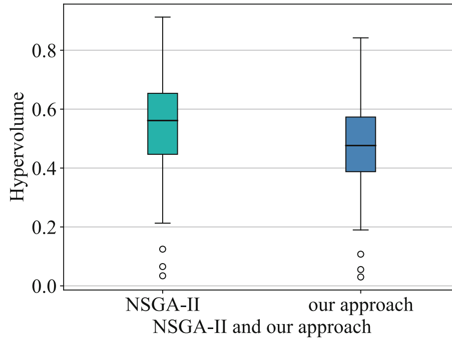

Fig. 2. The boxplot of HV values for two algorithms

In addition, to visually compare the performance of the two algorithms, Fig. 2 plots the HV values of the two algorithms in a box plot. A similar conclusion can be drawn from Fig. 2 that our approach performs better than NSGA-II (the box of our approach is lower, which indicates better results).

Answer to RQ1. By comparing our approach to NSGA-II, we demonstrate that our approach outperforms NSGA-II, which can achieve a 11.7% improvement in HV values. As a result, our approach can effectively explore the Pareto set and select better optimization sequences.

Investigation of RQ2. In this RQ, we attempt to investigate the impact of two parameters: crossover rate and mutation rate. A feasible set of parameters may result in preferable results with improved performance. To examine the impact of the crossover rate, we change its values while keeping the mutation rate fixed values, and vice-versa. Besides, the two parameters with varying probabilities are as follows: crossover rate = {0.5, 0.7, 0.9}, mutation rate = {0.01, 0.05, 0.1}. Similar to RQ1, the experiments are carried out on cBench and GCC.

Figure 3 depicts the impact results of these two parameters. We make the following three observations based on Fig. 3(a) and Fig. 3(b). First, our approach converges faster as the crossover rate increases. Second, we find that the convergence of our approach becomes slower when the mutation rate ranges from 0.01 to 0.1. As expected, the higher the crossover rate and the lower the mutation rate, the faster convergence occurs. Third, in terms of convergence rate, the experimental results of different parameter settings show no noticeable difference in crossover and mutation rates.

Answer to RQ2. The findings of the parameters impact analysis show that our approach is not very sensitive to the crossover and mutation rates that are set. In our experiment, we set the crossover rate to 0.9, and the mutation rate to 0.01.

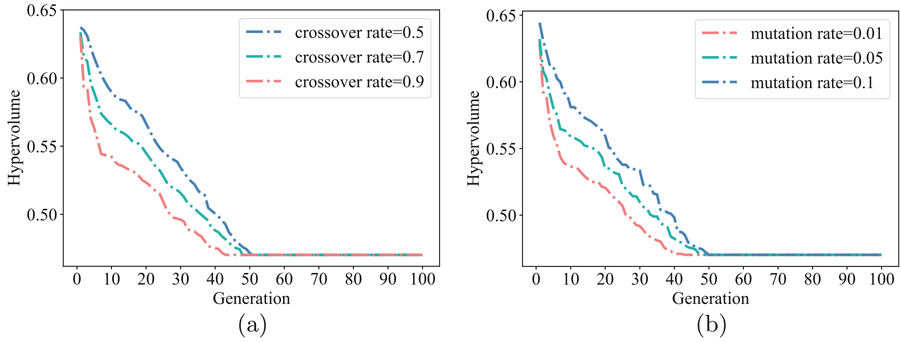

Fig. 3. The impact of crossover rate and mutation rate

Investigation of RQ3. To gain a better understanding of our approach, we investigate whether the two surrogate models can effectively improve the efficiency of our approach in RQ3. The number of actual fitness evaluations is applied as in [18] to investigate the value of surrogate models. By comparing the number of actual fitness evaluations required using our approach and a variant without surrogate models to reach convergence and stop the iteration, the results show that our approach converges when the number of actual fitness evaluations is 4400, while the variant requires 5600, i.e., more fitness evaluations. Besides, based on the conclusion of RQ1, we can conclude that our approach performs better and can find more suitable compiler optimization sequences while requiring fewer actual fitness evaluations.

Answer to RQ3. Taking into account the actual fitness evaluation of our approach and a variant without surrogate models, we conclude that the two surrogate models aid in improving the efficiency of our approach and make a contribution to solving the computationally expensive problem.

5 Conclusion and Future Work

In this paper, we present a novel surrogate-assisted multi-objective optimization algorithm to improve the efficiency of evolutionary algorithms for compiler optimization sequence selection. To address the diverse objectives influence and the computationally expensive problem, the proposed approach combines a fast global search based on non-dominated sorting with two random forests as surrogate models. The experimental results on cBench show that our proposed approach achieves better performance than NSGA-II and requires fewer actual fitness evaluations.

Despite the promising results, this work is still preliminary. Because of the complexity and variety of programs, different appropriate surrogate models will be designed for different programs in the future to improve efficiency. In addition, the dimensionality reduction method will be considered on account of the high-dimensional problem with a large number of compiler optimization options.

References

1. Agakov, F., et al.: Using machine learning to focus iterative optimization. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 295–305 (2006)
2. Ansel, J., et al.: OpenTuner: an extensible framework for program autotuning. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, pp. 303–316 (2014)
3. Ashouri, A.H., Bignoli, A., Palermo, G., Silvano, C., Kulkarni, S., Cavazos, J.: MICOMP: mitigating the compiler phase-ordering problem using optimization subsequences and machine learning. ACM Trans. Archit. Code Optim. **14**(3), 29 (2017)
4. Ashouri, A.H., Killian, W., Cavazos, J., Palermo, G., Silvano, C.: A survey on compiler autotuning using machine learning. ACM Comput. Surv. **51**(5), 1–42 (2018)
5. Ashouri, A.H., Mariani, G., Palermo, G., Park, E., Cavazos, J., Silvano, C.: COBAYN: compiler autotuning framework using Bayesian networks. ACM Trans. Archit. Code Optim. (TACO) **13**(2), 1–25 (2016)
6. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
7. Cáceres, L.P., Bischl, B., Stützle, T.: Evaluating random forest models for irace. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, pp. 1146–1153 (2017)
8. Chebolu, N.A.B.S., Wankar, R.: Multi-objective exploration for compiler optimizations and parameters. In: Murty, M.N., He, X., Chillarige, R.R., Weng, P. (eds.) MIWAI 2014. LNCS (LNAI), vol. 8875, pp. 23–34. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13365-2_3
9. Chen, J., Xu, N., Chen, P., Zhang, H.: Efficient compiler autotuning via Bayesian optimization. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 1198–1209. IEEE (2021)
10. Deb, K., Agrawal, R.B.: Simulated binary crossover for continuous search space. Complex Syst. **9**(2), 115–148 (1995)
11. Fursin, G.: Collective benchmark (cBench), a collection of open-source programs with multiple datasets assembled by the community to enable realistic benchmarking and research on program and architecture optimization (2010). http://cTuning.org/cbench
12. Gu, Q., Wang, D., Jiang, S., Xiong, N., Jin, Y.: An improved assisted evolutionary algorithm for data-driven mixed integer optimization based on Two_Arch. Comput. Ind. Eng. **159**, 107463 (2021)
13. Gu, Q., Wang, Q., Li, X., Li, X.: A surrogate-assisted multi-objective particle swarm optimization of expensive constrained combinatorial optimization problems. Knowl.-Based Syst. **223**, 107049 (2021)
14. Hall, M., Padua, D., Pingali, K.: Compiler research: the next 50 years. Commun. ACM **52**(2), 60–67 (2009)
15. Hong, W., Yang, P., Wang, Y., Tang, K.: Multi-objective magnitude-based pruning for latency-aware deep neural network compression. In: Bäck, T., et al. (eds.) PPSN 2020. LNCS, vol. 12269, pp. 470–483. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58112-1_32
16. Lokuciejewski, P., Plazar, S., Falk, H., Marwedel, P., Thiele, L.: Multi-objective exploration of compiler optimizations for real-time systems. In: 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, pp. 115–122. IEEE (2010)

17. Lokuciejewski, P., Plazar, S., Falk, H., Marwedel, P., Thiele, L.: Approximating pareto optimal compiler optimization sequences-a trade-off between WCET, ACET and code size. Softw. Pract. Experience **41**(12), 1437–1458 (2011)
18. Sun, C., Ding, J., Zeng, J., Jin, Y.: A fitness approximation assisted competitive swarm optimizer for large scale expensive optimization problems. Memetic Comput. **10**(2), 123–134 (2018)
19. Sun, Y., Wang, H., Xue, B., Jin, Y., Yen, G.G., Zhang, M.: Surrogate-assisted evolutionary deep learning using an end-to-end random forest-based performance predictor. IEEE Trans. Evol. Comput. **24**(2), 350–364 (2019)
20. Valdiviezo, H.C., Van Aelst, S.: Tree-based prediction on incomplete data using imputation or surrogate decisions. Inf. Sci. **311**, 163–181 (2015)
21. Zhou, Y.Q., Lin, N.W.: A study on optimizing execution time and code size in iterative compilation. In: 2012 Third International Conference on Innovations in Bio-Inspired Computing and Applications, pp. 104–109. IEEE (2012)
22. Zitzler, E., Thiele, L.: Multiobjective optimization using evolutionary algorithms—a comparative case study. In: Eiben, A.E., Bäck, T., Schoenauer, M., Schwefel, H.-P. (eds.) PPSN 1998. LNCS, vol. 1498, pp. 292–301. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0056872