SMARTEST: A Surrogate-Assisted Memetic Algorithm for Code Size Reduction

He Jiang, Guojun Gao, Zhilei Ren, Xin Chen, and Zhide Zhou

***Abstract*—Compiling source code effectively to meet various criteria is a critical task in software engineering. Especially, code size reduction has attracted much attention from both industry and academia due to the requirement of resource utilization. Generally, developers rely on compiler optimization passes to realize code size reduction. However, it is impractical to select a desirable optimization sequence manually since a wide variety of optimization passes are integrated into a compiler. Evolutionary algorithms offer an impressive way to alleviate this problem. Nevertheless, previous approaches fail to balance the exploitation and exploration of the search space. Moreover, the expensive fitness evaluation requires actual compilation, which makes the evolution rather time-consuming. To tackle the challenges, we propose a novel approach SMARTEST, which characterizes the systematic exploitation of a huge volume of historical compilation information. Specifically, SMARTEST comprises two components: 1) a local search operator to enhance the solution quality; and 2) a data-driven surrogate model to avoid expensive fitness evaluation. We evaluate the effectiveness of SMARTEST over the cBench benchmark suite. Experimental results indicate that SMARTEST outperforms the standard level -Os by 2.17% on average, and achieves 1.2 times code size reduction compared with the genetic algorithm. Furthermore, experimental results over the benchmark suite evidently show that SMARTEST gets a better result and takes less actual fitness evaluations than its variants, which demonstrates the contribution of the local search and the surrogate model.**

***Index Terms*—Code size reduction, compiler optimization selection, computationally expensive problems, memetic algorithm, surrogate-assisted.**

I. Introduction

In this new era, software application mediates almost every aspect of our lives. A lot of source code is produced every day, due to the increasing amount of applications. How to effectively compile source code to a target program to satisfy certain goals is a primary requirement of developers. By exploring the compilation solutions space and simulating the fault-injection, Serranocases *et al.* [1] apply a genetic algorithm (GA) and a multiobjective optimization approach to select some good optimization sequences to improve the reliability of programs. Besides, the size of executable files is also considered as a critical factor in the deployment of software applications, especially for embedded systems with limited on-chip memory space [2]–[4]. Most compilers provide the standard optimization level -Os to help developers optimize the code size of their programs. However, -Os may not meet the requirement for different programming languages, applications, and target architectures [2], [5], [6]. Besides, modern compilers (e.g., GCC[1] and LLVM[2]) offer an increasing number of optimization passes. For example, the compiler GCC has provided more than 200 compiler optimization passes. Even if we only consider whether an optimization pass is enabled or disabled, the size of the search space will exceed 2^{200}. This makes it unrealistic to select the best optimization sequence for target programs manually. Thus, developing advanced techniques is critical to help developers select better optimization sequences to optimize the code size of their programs.

So far, many techniques [3], [4], [7]–[10] have been proposed to automatically select compiler optimization passes for code size reduction. These techniques can be divided into two categories—namely, the machine learning-based approaches [3], [7] and the evolutionary algorithm-based approaches [4], [8]–[10]. The machine learning-based approaches focus on building machine learning models to predict the performance of different optimization sequences. In contrast, the evolutionary algorithm-based approaches often transform an optimization sequence (a set of optimization passes in this article) into a genetic representation and define a fitness function to determine the performance of the optimization sequence. During the evolution, it performs the mutation, the crossover, and the selection operators in an iterative paradigm. The machine learning-based approaches can quickly verify whether an optimization sequence can optimize the code size of a program, but it needs to construct a large dataset to train the prediction model. While the evolutionary algorithm-based approaches can better search the space to find an optimization sequence for reducing the code size

Manuscript received May 15, 2020; revised September 25, 2020 and January 18, 2021; accepted March 16, 2021. This work was supported by the National Natural Science Foundation of China under Grant 62032004, Grant 61722202, and Grant 61902096. Associate Editor: S. Liu. *(Corresponding author: He Jiang.)*

He Jiang is with the School of Software and DUT Artificial Intelligence Institute, Dalian University of Technology, and also with the Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, Dalian 116024, China (e-mail: jianghe@dlut.edu.cn).

Guojun Gao, Zhilei Ren, and Zhide Zhou are with the School of Software, Dalian University of Technology, Dalian, China (e-mail: ggj_gao@mail.dlut.edu.cn; zren@dlut.edu.cn; cszide@gmail.com).

Xin Chen is with the College of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou 310018, China (e-mail: chenxin4391@hdu.edu.cn).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TR.2021.3073960.

Digital Object Identifier 10.1109/TR.2021.3073960

[1] [Online]. Available: http://gcc.gnu.org/

[2] [Online]. Available: https://llvm.org/

of the target programs, but with a time-consuming procedure to determine the effectiveness of the optimization sequence. Thus, in this study, we focus on combining the advantages of these two approaches to further improve the code size reduction of programs.

Specifically, two major challenges need to be addressed when we utilize the evolutionary algorithm-based approaches effectively with machine learning techniques to find high-quality optimization sequences.

1) Neighborhood exploitation: The evolutionary algorithm-based approaches apply a global search algorithm like GA, which tends to concentrate more on the exploration of the search space and produce diverse optimization sequences. However, these approaches can fail to handle the exploitation mechanism properly, in that the neighborhoods of the current best individuals are not extensively investigated.
2) Expensive fitness evaluation: During the iterative process, the evolutionary algorithm-based approaches need to compile the program with all individuals to obtain their effectiveness. While this fitness evaluation process is very time-consuming.

To address the aforementioned challenges, we propose a novel approach surrogate-assisted Memetic AlgoRiThm for codE Size reducTion (SMARTEST) that efficiently selects promising optimization sequences for the program under compilation. First, to tackle the neighborhood exploitation challenge, we embed a local search procedure in the evolution process to improve the quality of each individual generated by the genetic operators. With a collaboration between the global search and the local search, a balance between the exploration and the exploitation could be achieved. Second, to tackle the expensive fitness evaluation challenge, we adopt a surrogate model to produce an approximate fitness score, instead of using the actual fitness score.

In particular, we train a random forests model to predict the fitness score of each individual. This prediction mechanism could alleviate the time-consuming issue of the fitness evaluation, which in turn accelerates the overall search procedure. By combining the local search and the surrogate model within GA, we obtain the integrated SMARTEST framework. Taking the compiler GCC as a case study, SMARTEST can achieve better optimization sequences, and reduce the computation of the optimization sequence evaluation.

To evaluate SMARTEST, we conduct experiments on the cBench [11] benchmark suite, which covers 32 programs including embedded functions and desktop programs. Experimental results demonstrate that our approach and GA perform better than the standard level -Os by 2.17% and 1.80% on average in terms of the code size reduction for programs. Besides, SMARTEST achieves 1.2 times code size reduction compared with GA. Furthermore, experimental results over the benchmark suite show that SMARTEST gets a better result and takes less actual fitness evaluations than its variants. It demonstrates that the local search and the surrogate model contribute to addressing the challenges of neighborhood exploitation and expensive fitness evaluation.

The contributions of this article are summarized as follows.

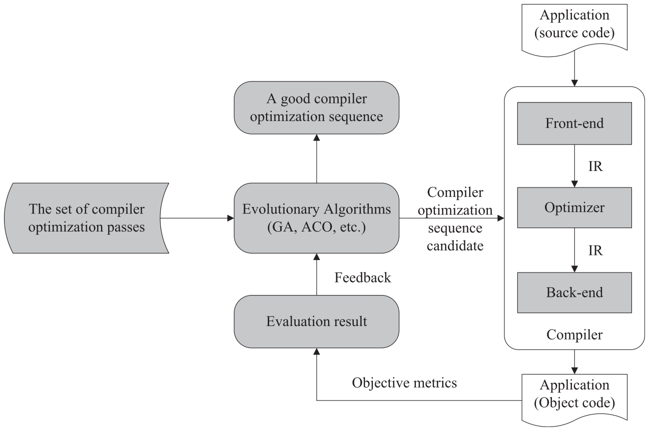

Fig. 1. Framework for selecting optimization sequences using evolutionary algorithms.

1) We propose SMARTEST, a novel approach to efficiently select optimization sequences to improve the code size reduction of programs.
2) We present a novel local search scheme to balance the exploitation and the exploration of the search space.
3) A surrogate model is proposed to mitigate expensive fitness evaluation. To the best of our knowledge, this is the *first* work to apply the surrogate model in selecting optimization sequences for code size reduction.
4) Extensive experiments conducted on the well-known compiler GCC and the cBench benchmark suite show that SMARTEST is effective. SMARTEST can select better optimization sequences than the standard level -Os and GA for a given program.

The rest of the article is structured as follows. We first provide the background and the motivation of our work in Section II and our approach in Section III. Then, we present the experimental setup, experimental results, and the threats to validity in Section IV. Next, we review related works in Section V. Finally, Section VI concludes this article.

II. Background and Motivation

A. Background

In this section, we present the background knowledge of selecting optimization sequences for code size reduction and demonstrate that it is a nontrivial task. An optimization sequence consists of a set of optimization passes. When we focus on whether an optimization pass is applied without regard to the ordering of these optimization passes, the problem is defined as *selecting optimization sequences*.

Fig. 1 shows a framework for selecting optimization sequences leveraging evolutionary algorithms. These algorithms define the set of compiler optimization passes as the search space and transform an optimization sequence into a genetic representation. Then, some optimization sequence candidates are selected to optimize the application under compilation. When the source code of an application is given as input of the compiler, the source code is transformed into an intermediate representation (IR) in the front-end. Then, a candidate of optimization sequences will be adopted to optimize the IR in optimizer. After that, the IR is transformed into the object code of the application in the back-end. An objective metric is usually defined in terms of performance, e.g., code size. Next, the framework evaluates these candidates of optimization sequences and guides evolutionary algorithms based on feedback information during the exploration. Finally, a good optimization sequence is generated.

The evolutionary algorithm-based approaches can effectively identify good optimization sequences that outperform the standard optimization levels [2], [4]. However, during the iterative process when evolutionary algorithms are applied, the compiler needs to compile the program with every optimization sequence to get the optimization performance result. Thus, all individuals are evaluated using the expensive, original fitness function. In fact, some valueless individuals are not necessarily evaluated using the original fitness function. Thus, it will lead to a computationally expensive problem and motivate us to consider a more profitable approach.

Formally, let seq be a Boolean vector, which denotes an optimization sequence. We use o_i to refer to the *i*th element of the vector seq, it indicates the *i*th compiler optimization pass applied in the sequence. An optimization pass o_i can be defined as a Boolean variable of which the value is either $o_i = 1$ (enabled), or $o_i = 0$ (disabled)

$$seq = (o_1, o_2, o_3, \ldots, o_n) \tag{1}$$

where n represents the number of optimization passes. These optimization passes are analyzed during the procedure of selecting optimization sequences. We can find that the search space is an exponential space (2^n). For example, when $n = 10$, there is a total of 1024 candidate sequences to be examined.

The compiler GCC is equipped with several standard levels (-O1, O2, -O3, and -Os) to help developers use predefined optimization passes. We take the version 7.3.0 as an example, -O1 turns ON 43 optimization passes and tries to help reduce code size and execution time, but those optimization passes that take a great deal of compilation time are not applied. -O2 optimizes even more than -O1 and turns ON all optimization passes specified by -O1 with 47 additional optimization passes. This level increases the compilation time and improves the performance of executable files. Besides, -O3 provides the most aggressive optimization passes, which turns ON all optimization passes specified by -O2 and 12 other optimization passes to optimize programs. -Os is designed to optimize for code size, which enables all -O2 optimization passes that do not increase code size.

In this study, we concentrate on reducing the compiled code size. For embedded systems or wireless sensor networks, optimizing for code size is a more noticeable problem, even though there is a tradeoff between the speed and the size. We consider 83 optimization passes of the compiler GCC 7.3.0 that the standard level -Os involved reducing code size. According to the definition of the above vector seq, the entire search space will become very large. Thus, selecting good optimization sequences by hand is unrealistic.

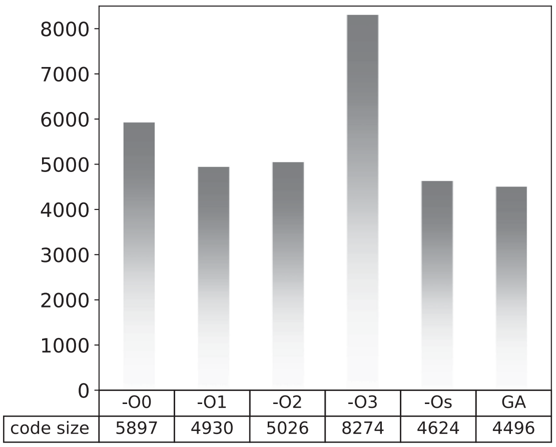

Fig. 2. Comparison of compiled code size among the standard levels and the sequence obtained by GA in *automotive_qsort1*.

B. Motivation

Previous studies [4], [9], [10] have attempted to apply evolutionary algorithms to select a better optimization sequence, such as GA. As an example, we select a sample program *automotive_qsort1* from the cBench to compare the compilation optimization results of these different optimization sequences, including the standard levels and the sequence obtained by GA.

Fig. 2 summarizes the code size of different executable files that the program *automotive_qsort1* is compiled using the standard levels (-O0, -O1, -O2, -O3, and -Os) and the optimization sequences generated by GA. From the figure, we can see that the code size is 5897 bytes using -O0. The other three standard optimization levels -O1, -O2, and -Os get 4930 , 5026 , and 4624 bytes, respectively. The result shows that the standard levels achieve significantly code size reduction over the default -O0 by about 15%–22% except for -O3. The reason is that -O3 is designed to optimize the performance, which may make the code size larger than -O0. Moreover, GA generates a better optimization sequence and gets a smaller code size (4496 bytes) than the standard levels. Therefore, the standard levels are not always effective enough, we need to choose the adaptive optimization sequence for each program.

Given a program P under compilation, we need to explore an optimization sequence that can benefit the code size of programs. We aim at finding an effective optimization sequence $\overline{seq} = \{o_{i1}, o_{i2}, o_{i3}, \ldots, o_{ij}\}$, where $1 \leq i1, i2, i3, \ldots, ij \leq n$. The evolutionary algorithm based approaches can effectively explore better optimization sequences over the standard levels. However, during the iterative search process in the previous studies, every individual is evaluated using the expensive fitness function. To evaluate an individual, the program is required to be compiled once with the optimization sequence. Thus, it leads to the computationally expensive problem.

In this article, we present a novel approach-based on the surrogate model to address this computationally expensive problem. When evaluating the optimization result of an optimization sequence in the local search, our approach applies the surrogate model to offer an approximate fitness score, instead of

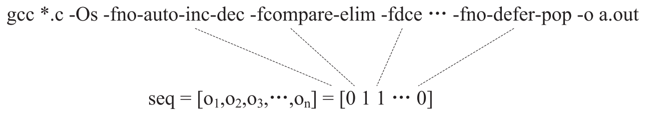
gcc *.c -Os -fno-auto-inc-dec -fcompare-elim -fdce ⋯ -fno-defer-pop -o a.out
seq = [o1,o2,o3,⋯,on] = [0 1 1 ⋯ 0]

Fig. 3. Example of solution representation.

compiling the program. In this way, our approach can resolve the computationally expensive problem.

III. Preliminaries

In this section, we first present the solution representation and the fitness function of selecting optimization sequences. Then, we describe the workflow of our approach, SMARTEST, followed by a more detailed description of its components, including the genetic operators, the local search, and the surrogate model.

A. Representation

Every individual in the population is called a candidate solution. In this study, a candidate solution is represented as an optimization sequence (a set of available optimization passes). We concentrate on whether an optimization pass is used. Thus, we use a Boolean vector, $seq = (o_1, o_2, o_3, \ldots, o_n)$, to represent an individual where each dimension is a compiler optimization pass. The values of the variables within the vector are 0 or 1, which are represented as genes in a chromosome. Eighty three optimization passes are enabled by the standard and the most aggressive level (-Os), i.e., $n = 83$. The full list of optimization passes we analyzed is available at the compiler GCC website[3]. Besides, the solution representation has a fixed length, which corresponds to the total number of the selected optimization passes. Because the order of optimization passes is fixed in GCC[4]. We do not need to consider the order of optimization passes in this article.

Fig. 3 shows an example of solution representation in this study. We set the variable o_i in the vector seq to 1 or 0, where 1 represents that we turn ON a specific optimization pass by using -f ⟨ optimization name ⟩, while 0 represents that we turn OFF the corresponding optimization pass using -fno-⟨ optimization name⟩. Thus, we use the method to control whether an optimization pass is applied or not. Then the program can be compiled using the optimization sequence that the individual represents. Furthermore, the compilation result of the optimization sequence on the program can also be evaluated.

B. Fitness Function

In evolutionary algorithms, the fitness score of a candidate solution is used to evaluate its quality. It also determines whether the candidate should be inherited to the next generation. We need to define a fitness function $Fitness(seq)$ that measures the code size reduction of different optimization sequences. In this study, we regard the code size as an optimization objective. Besides, we compare the performance of an optimization sequence with the default -O0 on the same program, because -O0 does not offer any optimization on the code size. Hence, this leads to the following fitness function:

$$Fitness(seq) = code\_size(\text{-O0}) - code\_size(seq) \quad (2)$$

where $code\_size(seq)$ is the code size of the object file after performing the optimization sequence seq on the program. According to the definition of the fitness function, it is to be maximized by the search.

Algorithm 1: SMARTEST.

```
Input: optimization passes $Optimizationpasses$, program
       $Program$
Output: the best optimization sequence $seq_{max}$ and the best
        fitness score $f_{max}$
begin
    $p_{num}, \alpha, \beta, N \longleftarrow$population size, crossover rate,
      elitism rate, maximum of iterations
    $Fitness() \longleftarrow$ Fitness function
    Initialize $population$
    Evaluate $population$ using $Fitness()$
    $f_{max} \longleftarrow$ Maximum fitness score
    $seq_{max} \longleftarrow$ Optimization sequence with maximum
      fitness score
    $generation \longleftarrow 1$
    Construct the surrogate model $Surrogate()$
    while $generation < N$ do
        number of elitisms $e_{num} = \beta p_{num}$
        Select the best $e_{num}$ individuals in $population$ and
          saved in $pop_a$
        // rest individuals for evolution
        $r_{num} = p_{num} - e_{num}$
        for $i \leftarrow 1$ to $r_{num}/2$ do
            // roulette wheel selection
            Select two individuals $seq_1$ and $seq_2$ from
              $population$ using the roulette wheel method
            // crossover operation
            if $random() < \alpha$ then
                $x_1, x_2 \longleftarrow$ crossover $seq_1, seq_2$
            else
                $x_1, x_2 \longleftarrow seq_1, seq_2$
            end
            Save $x_1, x_2$ in $pop_b$
        end
        // local search
        for $seq_i \in pop_b$ do
            local optimal individual $seq_i^{'} \longleftarrow$ Apply the
              local search in $seq_i$ using $Surrogate()$
            $seq_i \longleftarrow seq_i^{'}$
        end
        Evaluate $pop_b$ using $Fitness()$
        $population \longleftarrow pop_a \cup pop_b$
        Update the surrogate model $Surrogate()$
        Update the $f_{max}, seq_{max}$
        $generation \longleftarrow generation + 1$
    end
    return $seq_{max}, f_{max}$
end
```

C. Our Approach: SMARTEST

To resolve the challenges of the neighborhood exploitation and the expensive fitness evaluation as mentioned in Section I, we design a new evolution algorithm, named SMARTEST.

[3] [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc/Optimize-Options.html

[4] [Online]. Available: https://stackoverflow.com/questions/33117294/order-of-gcc-optimization-flags

In Algorithm 1, we present the pseudo-code of SMARTEST. Considering that a program will be compiled, we use SMARTEST to identify a good optimization sequence for reducing code size. First of all, we introduce several parameters, p_{num}, α, β, and N. Where p_{num} is the population size, α is the crossover rate, β is the elitism rate, and N is the maximum of iterations. Our approach first randomly generates the initial $population$ (line 4). The population size determines the convergence rate. The bigger the size is, the higher the convergence rate is. To evaluate its influence on SMARTEST, we experimentally set the initial population size in Section IV.

After the initialization, the program is compiled using these optimization sequences. An individual in the population represents one optimization sequence. Then we apply the fitness function $Fitness()$ to calculate the fitness scores of all individuals. By doing so, we can obtain and store the maximum fitness score f_{max} and the optimization sequence seq_{max} with the maximum fitness score (lines 5-7). Meanwhile, in line 9, SMARTEST constructs the surrogate model $Surrogate()$ using the compilation information in the dataset, including the optimization sequences in the initial population and their effectiveness in terms of the code size for the target program.

During each iteration (lines 10-32), we first select and store a number of best individuals pop_a of the population, which ensures that these individuals automatically survive in evolution (lines 11-12). For the rest of the individuals in the current population (lines 14-26), the selection, the crossover, and the local search will be employed to produce new individuals in order. At the beginning of this process, the roulette wheel selection is utilized to select two parents from the current population (line 15). It makes the genes of the sterling individuals pass on to the next generation with a higher probability. Then, we use the single-point crossover operator to generate new individuals based on the crossover probability α (lines 16-21). After the crossover operation, via using the surrogate model $Surrogate()$, the local search is applied to search for the local optimum of the individuals in the rest population pop_b. Once obtaining its local optimum, all individuals are substituted with their corresponding local optimum (lines 23-26).

Following these three operations, in line 27, we adopt the fitness function to re-evaluate these newly generated individuals in pop_b. After that, we combine the elite individuals pop_a and the new individuals pop_b to form the next population (line 28). When the next population is generated, we added the new compilation information to the dataset and update the surrogate model $Surrogate()$, the maximum fitness score f_{max}, and the optimization sequence seq_{max} with the maximum fitness score (lines 29-30).

This iterative process repeats until a predefined number of generations N has been reached. Finally, we get the result with the best optimization sequence seq_{max} and its fitness score f_{max}.

D. Genetic Operators

In this work, we perform two genetic operators on the population: the selection and the crossover. These two operators are applied to help drive the algorithm toward obtaining the best optimization sequence.

1) Selection: The goal of the selection is to determine which genes of candidates should be carried to the next generation. In our approach, we would like to favor the individuals that lead to a larger code size reduction with a higher fitness score. The roulette wheel selection is adopted as the selection mechanism for this purpose [12], [13].

In the roulette wheel selection, the probability of selecting the individual seq_i is $Fitness(seq_i)/\sum_{i=1}^{p_{\text{num}}} Fitness(seq_i)$, where $Fitness(seq_i)$ is the fitness score of the individual seq_i and p_{num} is the total number of individuals in the current population. Therefore, the higher the fitness score of an individual is, the more chances it is to be selected. For example, the population contains 4 individuals and their fitness scores are $\{1, 2, 3, 4\}$, respectively. Summing these fitness scores, we can apportion a percentage of total fitness. It gives the strongest individual of a value of 40% and the weakest 10%. This percentage of fitness scores can be used to configure the roulette wheel. Thus, the probabilities of selecting these four individuals are $\{10\%, 20\%, 30\%, 40\%\}$. Each time the wheel stops, the fitter individual has a larger chance of being selected.

2) Crossover: By applying the crossover operator [14], we can combine the genetic information of two selected parents to generate new offspring. It is adopted to evolve high-quality individuals from the existing population and form a new population. Since the chromosome (in our case, it stores the genetic information of the optimization sequence) is represented by a binary array, we use the single-point crossover [15] to support the recombination.

For the single-point crossover, firstly, a single crossover point is randomly selected in the parents' sequences. Then, two new offsprings are produced by exchanging the genetic information after the selected random position. Newly generated solutions are added to the new population by applying the crossover operator to all parents. For example, two individuals $i = (1010)$ and $j = (1101)$ are selected as parents, after applying the single-point crossover operator in position two, their offsprings will be (1001) and (1110).

E. Local Search

As discussed in Section I, a major challenge faced by the evolutionary algorithm-based approaches lies in the lack of the exploitation of intensification. Hence, we apply a local search operator to systematically explore the immediate neighborhood of each incumbent individual, in search of opportunity for local improvement. Thus, SMARTEST can potentially find a better optimization sequence.

We detail the local search procedure in Algorithm 2. For an individual seq_i in the current population, we produce candidate neighbors $N(seq_i)$ of seq_i by flipping each bit first (line 4). The number of candidate neighbors is equal to the population size. Then all these neighbors are evaluated with respect to their fitness (line 5). After selecting the best neighbor with the maximum fitness (line 6), the search continues until the local optimum seq_i' is found. Thus, we replace the original individual with the local optimum in the current population. During every repetition process of finding the local optimum, we only consider the candidate neighbors of the individual, if the individual

Algorithm 2: Local Search.

```
Input: individual $seq_i$
Output: local optimal individual $seq_i^{'}$
begin
    while true do
        $seq_i^{'} \longleftarrow seq_i$
        // Generate candidate neighbors
        Generate($N(seq_i)$)
        Evaluate $N(seq_i)$
        $seq_i \longleftarrow \arg\max(N(seq_i))$
        if $seq_i = seq_i^{'}$ then
            break
        end
    end
end
return $seq_i^{'}$
```

has $N(seq_i)$ bits, we consider $N(seq_i)$ candidate neighbors. Thus, it will help us to systematically explore the immediate neighborhood and ensure the result is still local optimum. By embedding the local search in each generation of the evolution, SMARTEST is essentially a memetic algorithm [16].

F. Surrogate Model

To alleviate the time-consuming issue of the fitness evaluation, we employ a surrogate model in SMARTEST. The design of the surrogate model is based on the random forests model [17], which is one of the most commonly used surrogate models [18]–[21]. Random forests are an ensemble learning technique, in which several decision trees are organized as a whole model for classification or regression tasks. By generating many decision trees at training time and aggregating their results, the method outputs the voting result or the mean prediction of all trees. Compared with an individual tree, random forests can obtain better performance by correcting over-fitting to the training set.

In this study, we utilize the random forests model as the surrogate model. During the execution of SMARTEST, we first randomly generate many initial individuals and evaluate their effectiveness in terms of the code size for the target program. Then, the initial individuals are treated as features, and the effectiveness of these individuals is their labels. Each feature value in individuals is 1 or 0, indicating whether the optimization pass is used or not. We train the initial surrogate model using these initial individuals (lines 4 to 9 in Algorithm 1). In the iterative process, the individuals in each iteration are added to retrain the surrogate model to obtain a more accurate prediction (line 29 in Algorithm 1). The random forests model works as follows.

1) Repeatedly select a random sample with replacement of the original training set for N times for each tree construction. Thus, $Ntree$ bootstrap samples are drawn.
2) For each of the bootstrap samples, build an unpruned tree. Then, at each node, choose the best split from $Mtry$ features that are randomly sampled. The tree is built until no splits are possible or the node size reaches.
3) Predict a new sample by aggregating the prediction results of all the trees, i.e., averaging the predictions of multiple regression trees.

Where $Ntree$ is the number of trees in the forest, $Mtry$ is the number of features that are randomly selected for all splits.

Our approach builds a regression tree using randomly selected training samples. Some samples are left out, which are used to test the accuracy of the tree. Here, we apply RandomForestRegressor in Python to implement our model, and default parameters are used. For example, the number of trees in the forest is 100, the maximum depth of the tree is set none. The details of parameter settings are listed on the official web site[5].

We apply root-mean-square error (rmse) to estimate the error rate of the random forests model. The rmse is calculated as shown below, where y is the actual value, and $\dot{y}$ is the predicted value. The estimation of the error rate will guarantee and help us build a good random forests model

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{n}(\dot{y} - y)^2}. \tag{3}$$

Once the model is constructed, we use the model to predict the reduced code size of object code performed on an optimization sequence, instead of the actual value. It can solve the challenge of expensive fitness evaluation by reducing the actual fitness evaluation number.

The procedure is beneficial for obtaining better predictive performance by decreasing the bias, because a single tree may be highly sensitive to noise in the original training set. However, simply training several trees on the same training set are correlated, bootstrap sampling is an appropriate method to train different de-correlated trees.

IV. Experiments and Results

A. Experimental Setup

In this section, we experimentally evaluate the proposed SMARTEST. Prior to the result presentation of our evaluation, we first introduce the experiment design and the research questions (RQs). All the experiments are repeated 15 times with different random seeds since the evolutionary algorithms have the stochastic nature. Since the objective under consideration is the code size, according to the definition of the fitness function introduced in Section III-B, we need to obtain the code size of the executable file. In this study, we use the *Size* command to get the sum of the size of the text and data segment as the code size in terms of bytes. The text segment, also known as the code segment, is a portion of an object file. It corresponds to the program's virtual address space that contains executable instructions. Then, the data segment is also a portion of an object file and corresponds to the address space of a program that contains initialized static variables. Besides, SMARTEST is written in Python, because of the large availability of libraries. Then, all the experiments are conducted on an Intel Core i7 3.60 GHz CPU with 8 GB memory, running GNU/Linux with kernel 4.15.0.

To systematically evaluate SMARTEST, we investigate and answer the following RQs.

[5] [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

RQ1: How do the parameters impact SMARTEST?

Motivation: When applying SMARTEST to select good optimization sequences, different parameters need to be chosen, including the population size, the crossover rate, and the elitism rate. The choice of these three parameters might have an impact on the performance of SMARTEST. A better parameter setting might lead to a good optimization sequence with better performance. To shed light on the problem of parameters, we design this RQ.

Method: We analyze the impact of parameters on SMARTEST and tune parameters to obtain faster convergence in RQ1, including the population size, the crossover rate, and the elitism rate. The first parameter is the population size, which determines how many individuals are created in the population. The second parameter is the crossover rate. It specifies the probability of whether two individuals are crossed over, if not, these two individuals will be passed on to the next generation. The third parameter is the elitism rate, which determines how many the best individuals (i.e., elites in population) are copied to the next generation from the current population without any change. To shed light on the impact of these three parameters, we perform an empirical analysis on the cBench benchmark suite and choose the best configuration.

RQ2: How effective is SMARTEST?

Motivation: The most important criterion for SMARTEST to be effective is that it should be able to find competitive optimization sequences. In this RQ, we would like to investigate the quality of the optimization sequence selected by SMARTEST, and whether it is superior to GA and the standard level -Os.

Method: In this article, we explore the optimization sequence with the compiler GCC for our research, because it is a popular open-source compiler that supports many languages and architectures, it is also equipped with a large number of program optimization techniques. Furthermore, we compile all programs with the compiler GCC on the cBench [11], [22] benchmark suite and evaluate our approach, which contains a wide variety of programs ranging from embedded functions to desktop programs and is commonly adopted in finding optimization sequences [23]–[26].

We compare the performance of SMARTEST, GA_s, and GA in terms of the average best fitness reached among all the runs. As a baseline algorithm, GA in this experiment represents a variant of SMARTEST in which neither the local search nor the surrogate model is considered. Besides, GA_s represents a standard GA that contains a mutation operator. These algorithms, either our approach or the baseline algorithms, consist of 100 generations, which is consistent with existing guidelines [5], [9], [27], [28].

RQ3: How do the local search and the surrogate model contribute to the optimization sequence selection?

Motivation: In RQ2, we have demonstrated the effectiveness of SMARTEST as an overall framework. To gain more insights into the framework, we are interested in how the local search and the surrogate model collaborate to achieve promising results.

Method: In this RQ, we consider SMARTEST and its two variants, i.e., the variant without the surrogate model (SMARTEST-s), and the variant without the surrogate model and the local search (SMARTEST-s-l). The experimental setup is the same as RQ2. Then, we apply the variants and experiment on the cBench benchmark suite again. By comparing the original SMARTEST, SMARTEST-s, and SMARTEST-s-l, we examine whether the local search helps to tackle the challenge of neighborhood exploitation and whether the surrogate model is helpful to resolve the challenge of expensive fitness evaluation.

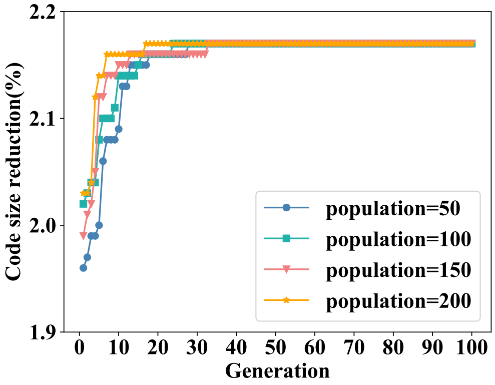

Fig. 4. Impact of the population size.

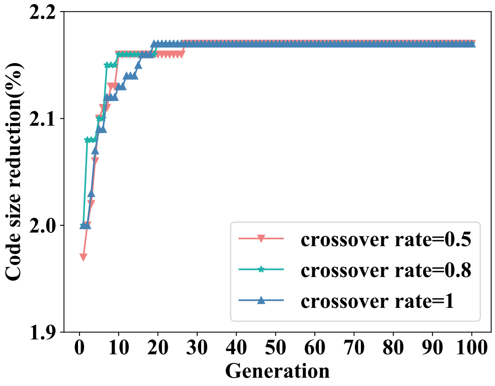

Fig. 5. Impact of the crossover rate.

B. *Investigation of RQ1*

In this RQ, we intend to investigate the impact of the three parameters on SMARTEST using the convergence rate, including the population size, the crossover rate, and the elitism rate. To evaluate possible configurations of these parameters, we opt for a set of values that are commonly used in search-based software engineering (SBSE) problems [29]. Figs. 4– 6 provide the impact analysis results of these three parameters. When we analyze the impact of one parameter, only the parameter is changing, the other two parameters keep the fixed values. In the experiment design, we set the default values of these three parameters: population size = 100, crossover rate = 0.8, and elitism rate = 0.1. Besides, the various parameters with different probabilities are described as follows:

1) population size: {50, 100, 150, 200};
2) crossover rate: {0.5, 0.8, 1};
3) elitism rate: {0.01, 0.1, 0.2, 0.5}.

In addition, we conduct a paired T-test to explore the statistical significance of the difference between different probabilities for three parameters. In this study, the significance level is set to

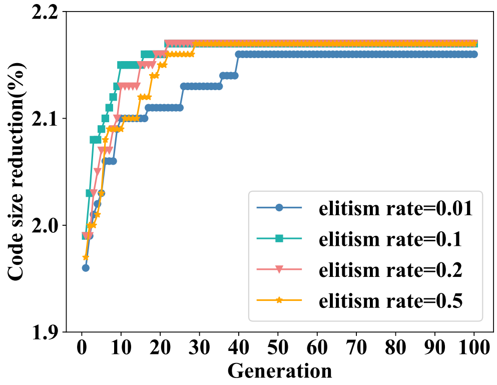

Fig. 6. Impact of the elitism rate.

0.05. If the p-value is less than the significance level, there is a significant difference between the two probabilities.

1) Population Size: We first consider the impact of the population size, which represents the number of optimization sequences presented in the population. To evaluate the impact of the population size, we examine the number of generations that SMARTEST takes to converge for different population sizes ranging from 50 to 200 (i.e., 50, 100, 150, 200). The crossover rate and the elitism rate are 0.8 and 0.1, respectively. Fig. 4 shows the impact of the population size. From the figure, we can find that the larger population size attains convergence faster. For example, with increasing population size from 50 to 200, the number of generations taken to arrive at a candidate solution becomes less. Moreover, the p-values are 0.3954, 0.2503, 0.3219, 0.7127, 0.1090, and 0.2274 among four population sizes. The p-values are higher than 0.05. It means that different population sizes have no significant difference. Notice that apart from population size = 50, the difference among the other three values is very negligible. However, increasing the population size will lead to consuming more time. Hence, we restrict the population size to 100 for SMARTEST.

2) Crossover Rate: As mentioned in Section III-D2, the crossover rate is applied to determine the probability of whether two individuals are crossed over. According to the guidance of SBSE [29], we investigate three different crossover rates: {0.5, 0.8, 1}. Fig. 5 shows the impact of varied crossover rates. From the figure, the following observations can be drawn. First, with the increasing of crossover rate, the algorithm will achieve convergence faster, because a larger crossover rate is associated with more exploration. Second, even though the algorithm achieves convergence differently, the final reduced code size is the same (2.17%) with different crossover rates. In addition, the p-values are 0.3577, 0.9336, and 0.2922 among the three crossover rates. These three p-values are higher than 0.05. It means that there is no significant difference among the three crossover rates. It demonstrates that SMARTEST is not very sensitive to the parameter of the crossover rate when it is applied to find a good optimization sequence for code size reduction.

From the result of Fig. 5, we can find that SMARTEST can get better results when the crossover rate is 0.8 in the early stage of algorithm running, especially in the first 20 generations. Besides, a higher crossover rate combined with local search can efficiently explore more of the possible space of solutions and avoid falling into the local optimal solution. Therefore, we choose a crossover rate of 0.8 for our approach.

3) Elitism Rate: The elitism rate determines how many individuals can pass on to the next generation straightforwardly. Fig. 6 provides the results obtained from the different elitism rate values: {0.01, 0.1, 0.2, 0.5}. From the figure, we can see that the approach may not achieve the candidate solution when the elitism rate = 0.01 (i.e., elitism number = 1). Besides, the results indicate that SMARTEST finds the candidate solution within the shortest generations when the elitism rate is 0.1. In addition, the p-values are 1E-07, 2.9E-05, and 0.0021 between 0.01 and the other three rates. While the p-values are 0.4064, 0.0825, and 0.3664 between the biggest three rates. The p-values are higher than 0.05 when the elitism rate is not 0.01. It means that there is no significant difference among the biggest three rates. Hence, we treat 0.1 as the parameter setting of the elitism rate.

Answer to RQ1: Experimental results show that, SMARTEST is not very sensitive to the above three parameters. We choose the parameter setting that performs relatively well, i.e., population size = 100, crossover rate = 0.8, and elitism rate = 0.1.

C. Investigation of RQ2

In RQ2, we investigate whether SMARTEST can effectively identify good optimization sequences for code size reduction. As the recommended parameter setting in RQ1, the population size is 100, the crossover rate is 0.8, and the elitism rate is 0.1. Besides, for GA_s(the standard GA, which contains the mutation operator), we use the mutation rate of 0.05 per gene [30]. Table I shows the code size reduced for each program with different optimization sequences, including -O0, -Os, and the optimization sequences generated using GA and SMARTEST. The first column represents the programs in the cBench benchmark suite we analyzed in this study. The columns four, six, and eight present the average value for the 15 executions on each program for GA, GA_s, and SMARTEST. An exact number for each program is reported on the columns two and three that indicate the code size of executable files are compiled with -O0 and -Os, respectively. The smaller the code size value, the better the optimization sequence generated by the corresponding method is. Additionally, we also report the code size reduction and the percentage of code size reduction over the standard level -Os for GA, GA_s, and SMARTEST in columns five, seven, and nine of Table I. Thus, the higher value is to be preferred. Note that the values in these columns are averaged over all programs, as shown at the bottom of the table.

It can be seen from Table I that SMARTEST and GA outperform the standard default optimization level for most programs of the benchmark suite. When applying GA to find good optimization sequences for code size reduction, GA achieves 0.04% to 5.58% code size reduction with an average value of 1.80%. Besides, when SMARTEST is utilized, it achieves 2.17% of reduction on average and can identify a better optimization sequence than -Os. Compared with GA, SMARTEST can even obtain more code size reduction. The results, as shown in Table I, indicate that SMARTEST is more effective than GA for code size reduction.

TABLE I
NUMBERS ON COLUMNS 2 AND 3 ARE CODE SIZE (BYTES) FOR -O0 AND -OS

program	-O0	-Os	GA	GA-Os	GA_s	GA_s-Os	SMARTEST	SMARTEST-Os
automotive_bitcount	8008	6731	6655	76(1.13%)	6655	76(1.13%)	6655	76(1.13%)
automotive_qsort1	5897	4624	4496	128(2.77%)	4496	128(2.77%)	4496	128(2.77%)
automotive_susan_c	45142	26280	25266	1014(3.86%)	25266	1014(3.86%)	25266	1014(3.86%)
automotive_susan_e	45142	26280	25234	1046(3.98%)	25234	1046(3.98%)	25234	1046(3.98%)
automotive_susan_s	45142	26280	25234	1046(3.98%)	25234	1046(3.98%)	25234	1046(3.98%)
bzip2d	126613	66072	65023	1049(1.59%)	64927	1145(1.73%)	64807	1265(1.91%)
bzip2e	126613	66072	65023	1049(1.59%)	64927	1145(1.73%)	64807	1265(1.91%)
consumer_jpeg_c	207629	127414	125766	1648(1.29%)	125774	1640(1.29%)	125694	1720(1.35%)
consumer_jpeg_d	200128	122587	120136	2451(2.00%)	120931	1656(1.35%)	119282	3305(2.70%)
consumer_lame	301024	203575	201535	2040(1.00%)	202420	1155(0.57%)	199515	4060(1.99%)
consumer_mad	290693	229769	227458	2311(1.01%)	227610	2159(0.94%)	227248	2521(1.10%)
consumer_tiff2bw	335390	249779	248500	1279(0.51%)	248550	1229(0.49%)	246445	3334(1.33%)
consumer_tiff2rgba	332780	248497	247566	931(0.37%)	247565	932(0.38%)	245125	3372(1.36%)
consumer_tiffdither	333526	249235	248075	1160(0.47%)	248165	1070(0.43%)	247859	1376(0.55%)
consumer_tiffmedian	340614	253091	251814	1277(0.50%)	251724	1367(0.54%)	251451	1640(0.65%)
network_dijkstra	5534	4839	4726	113(2.34%)	4736	103(2.13%)	4684	155(3.20%)
network_patricia	7834	5152	5123	29(0.56%)	5142	10(0.19%)	5102	50(0.97%)
office_ispell	82806	58563	57595	968(1.65%)	57652	911(1.56%)	57451	1112(1.90%)
office_ghostscript	1584243	1082501	1060850	21651(2.00%)	1063250	19251(1.78%)	1050025	32476(3.00%)
office_rsynth	148415	133783	133101	682(0.51%)	133005	778(0.58%)	132933	850(0.64%)
office_stringsearch1	9127	7592	7168	424(5.58%)	7163	429(5.65%)	7152	440(5.80%)
security_blowfish_d	23253	17677	17162	515(2.91%)	17162	515(2.91%)	17146	531(3.00%)
security_blowfish_e	23253	17677	17162	515(2.91%)	17162	515(2.91%)	17146	531(3.00%)
security_pgp_d	261359	191935	189752	2183(1.14%)	189752	2183(1.14%)	189472	2463(1.28%)
security_pgp_e	261359	191935	189752	2183(1.14%)	189752	2183(1.14%)	189472	2463(1.28%)
security_rijndael_d	71646	55830	55806	24(0.04%)	55806	24(0.04%)	55742	88(0.16%)
security_rijndael_e	71646	55830	55806	24(0.04%)	55806	24(0.04%)	55742	88(0.16%)
security_sha	6619	5067	4867	200(3.95%)	4858	209(4.12%)	4835	232(4.58%)
telecom_adpcm_c	4718	4467	4352	115(2.57%)	4328	139(3.11%)	4294	173(3.87%)
telecom_adpcm_d	4718	4467	4352	115(2.57%)	4328	139(3.11%)	4294	173(3.87%)
telecom_CRC32	5949	5801	5769	32(0.55%)	5769	32(0.55%)	5769	32(0.55%)
telecom_gsm	73796	52089	51568	521(1.00%)	51568	521(1.00%)	51302	787(1.51%)
average	168457	118797	117272	1525(1.80%)	117397	1399(1.79%)	116615	2182(2.17%)

The Numbers on Columns 4, 6, and 8 are the Average Code size(bytes) of the 15 Executions for GA, GA_s(standard GA, Which Includes Mutation Operator), and SMARTEST. Columns 5, 7, and 9 are A(B%): (A)code Size Reduction and (B)percentage Code Size Reduction W.r.t. Optimized Code by -Os

Taking the program *office_stringsearch1* as an example. The code size of executable files is 9127, 7592, 7168, and 7152 that are compiled with the unoptimized, the standard level -Os, GA, and SMARTEST, respectively. For the task of code size reduction, both SMARTEST and GA can find good optimization sequences to generate smaller executable files than -Os. Besides, GA can achieve a 5.58% code size reduction over -Os, while SMARTEST obtains a better reduction, by 5.80%. This observation confirms the results from previous studies [8], [9] that evolutionary algorithms can achieve better optimization sequences than the standard levels. What is striking about the results on most programs is that SMARTEST performs better than GA. This is likely due to the fact that SMARTEST employs two helpful components, i.e., the surrogate model and the local search. The surrogate helps SMARTEST search for more sequences within the same amount of time. The local search can find better individuals in the population.

Besides, we can find that the mutation can also improve the performance of the approach on some programs, such as *bzip2d*, *office_rsynth*, *security_sha*, and *telecom_adpcm_c*. Taking the program *bzip2d* as an example, GA_s applies mutation and achieves 1.73% code size reduction, while GA is 1.59%. However, according to the result of the paired T-test, the p-value is 0.7569 between GA and GA_s and is higher than 0.05. Thus, we think that GA and GA_s have no significant difference in the entire benchmarks. In addition, on most programs, GA_s and GA get the same code size reduction. The potential reason may be that the relation among different compiler optimization passes is very complex and the distribution of good optimization sequences is scattered, the mutation operation may not help GA generate better compiler optimization sequences efficiently.

From the table, we can observe several interesting phenomena. First, the results of SMARTEST are approximately the same as GA on some programs, such as automotive_bitcount, automotive_qsort1, and telecom_CRC32. This was probably because these programs can hardly be reduced or the opportunity of code size reduction is very small. Thus, SMARTEST cannot find a better chance than GA to reduce the executable file. Second, the experiment empirically confirms that the proposed approach, SMARTEST, can effectively find better optimization sequences than GA since it achieves more reduction than GA on about 80% of the programs.

In addition, we conduct a paired T-test to explore the statistical significance of the difference between three approaches, including GA, GA_s, and SMARTEST. The p-values are 1.49099E-05 between SMARTEST and GA, 3.07269E-05 between SMARTEST and GA_s, and 0.7569 between GA and GA_s. The p-values between SMARTEST and the other two approaches are lower than 0.05. It means that there is a significant difference between SMARTEST and other approaches in

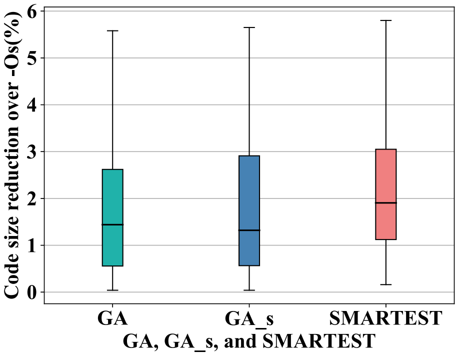

Fig. 7. Boxplot of code size reduction over -Os among SMARTEST, GA and GA_s.

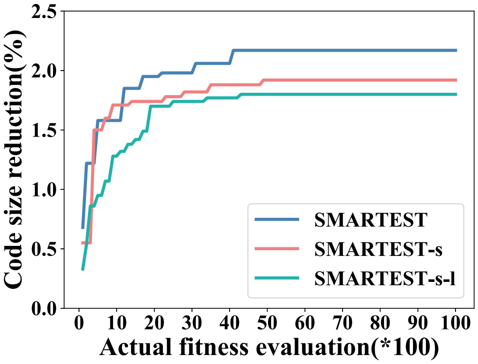

Fig. 8. Comparison between SMARTEST, SMARTEST-s, and SMARTEST-s-l.

a pairwise comparison. While the p-value of GA and GA_s is higher than 0.05, it indicates that these two approaches have no significant difference.

Moreover, we provide the boxplot in Fig. 7 to show descriptive statistics for GA, GA_s, and SMARTEST according to the code size reduction over -Os. The boxplot for the two approaches shows the median value as the centerline. Then, the lower and upper hinges of each box represent 25% and 75% quantile, respectively. The lower and upper whiskers indicate the smallest and largest values, respectively. As seen from the figure, SMARTEST, GA_s, and GA can generate smaller executable files than -Os (all reduction values are greater than 0), which demonstrates that only using the standard levels is not enough. In addition, SMARTEST performs better than GA and GA_s as a whole and can effectively find a better optimization sequence for code size reduction (the box of SMARTEST is higher).

Answer to RQ2: By comparing SMARTEST with the baseline GA and GA_s, as well as the standard level -Os, we demonstrate that SMARTEST, GA_s, and GA can identify good optimization sequences. All three approaches can generate smaller executable files than -Os. Furthermore, SMARTEST can effectively find better optimization sequences than GA and GA_s.

D. *Investigation of RQ3*

In this RQ, we validate whether the local search and the surrogate model help tackle the two challenges in selecting optimization sequences. On all programs in the cBench benchmark suite, we compare the performance of code size reduction, the actual number of fitness evaluations, and average execution time of SMARTEST, SMARTEST-s, and SMARTEST-s-l. In this study, we use the actual fitness evaluations to represent the execution time of the algorithms as in [31], since the execution time of each iteration is different and the actual evaluation takes up most of the execution time of the three algorithms. We record the code size reduction every one hundred actual fitness evaluations and continue until 10 000 evaluations. Fig. 8 shows the impact on the local search and the surrogate model for programs with the change in the actual number of fitness evaluations.

As shown in Fig. 8, compared with SMARTEST-s, SMARTEST-s-l converges faster, which indicates that SMARTEST-s-l gets the final optimization sequence using less actual fitness evaluations. However, SMARTEST-s obtains better optimization sequences than SMARTEST-s-l at the end of the evolution. A possible reason may be that SMARTEST-s-l focuses more on the diversification mechanism. In contrast, SMARTEST-s provides better intensification via the local search operator and can exploit the neighborhood structure of the individuals more effectively. Thus, it is demonstrated that the local search is beneficial for solving the challenge of neighborhood exploitation.

Furthermore, we compare SMARTEST with SMARTEST-s for analyzing the contribution of the surrogate model. By comparing the curves associated with the two variants, similar observations could be obtained, i.e., SMARTEST converges faster than SMARTEST-s, and SMARTEST finds better compiler optimization sequences when the iteration terminates. The potential reason for this finding is that SMARTEST-s puts a lot of effort into local search to search for the local optimum, there are many low-quality solutions that are actually compiled. In contrast, the surrogate model can drive SMARTEST to increase the opportunity to explore more search space, which is useful for finding more optimal solutions and prevents search from being trapped into local optimum. Similar conclusions are also obtained by Zhong *et al.* [32]. Therefore, SMARTEST tends to explore search spaces much larger than SMARTEST-s and has more global search ability to explore new search areas. Therefore, the surrogate model is substantially helpful in solving expensive fitness evaluation challenge.

Besides, we also report the details of the code size reduction and the actual fitness evaluations for these three variants to converge in Table II. From the table, we find that SMARTEST can achieve 2.17% of code size reduction when the number of actual fitness evaluations is 4100. While SMARTEST-s and SMARTEST-s-l find worse optimization sequences after 4900 and 4300 actual fitness evaluations, which reduces 1.92% and 1.80% of code size, respectively. Then, the average execution time of the three approaches for each program is listed in Table II. We can find that SMARTEST takes an average of 6490 seconds on each program, of which 208 seconds are used to train the surrogate model. While SMARTEST-s and SMARTEST-s-l take 7508 and 6588 s, respectively. Compared with SMARTEST-s-l,

TABLE II
CODE SIZE REDUCTION, THE ACTUAL FITNESS EVALUATIONS, AND AVERAGE EXECUTION TIME WHEN THREE APPROACHES CONVERGE

Approaches	code size reduction	actual fitness evaluations	average execution time(s)
SMARTEST	2.17%	4100	6490
SMARTEST-s	1.92%	4900	7508
SMARTEST-s-l	1.80%	4300	6588

although SMARTEST needs to spend extra time to train the surrogate model, it can still find better compiler optimization sequence to achieve code size reduction within less time. It demonstrates that it is valuable to spend some time training the surrogate model.

In addition, we conduct a paired T-test to explore the statistical significance of the difference between three approaches, including SMARTEST, SMARTEST-s, and SMARTEST-s-l. The p-values are 9E-05 between SMARTEST and SMARTEST-s, 3.11E-06 between SMARTEST-s and SMARTEST-s-l, and 1.81E-16 between SMARTEST and SMARTEST-s-l. The p-values are lower than 0.05. It means that there is a significant difference between SMARTEST and other approaches in a pairwise comparison.

Therefore, by employing the local search and the surrogate model, SMARTEST can identify befitting optimization sequences and consumes less actual fitness evaluations than SMARTEST-s and SMARTEST-s-l. This offers a reasonable way to address the two challenges identified in this study.

Answer to RQ3: Comparing the results of code size reduction and actual fitness evaluation of SMARTEST with its variants, we conclude that the local search is advantageous to address the challenge of neighborhood exploitation. Furthermore, the surrogate model is indeed helpful in addressing the challenge of expensive fitness evaluation.

E. Threats to Validity

In this section, we discuss some of the potential threats to the validity of this study.

1) Internal Validity: Threats to internal validity might come from the possible faults in our algorithm implementation. To mitigate this threat, we reviewed all the source code and tested the implementation carefully before the experiments were conducted. Furthermore, the performance of the search algorithm may vary with the random variations in the approach. To cope with this problem, we followed the guideline in SBSE [29] and ran the approach 15 times with different random seeds.

2) External Validity: With respect to external validity, we collect 32 programs in the cBench benchmark suite from prior work [23]–[26]. Although these programs belong to several types, including embedded systems and desktop programs, they may not be representative and not enough in general. More research needs to be conducted to evaluate the performance of our approach. Besides, we only consider the code size as the optimization objective in our study. However, there are other optimization performance metrics, such as the execution time and the energy efficiency, we hence strongly encourage the application of our approach to these metrics.

V. RELATED WORK

This section reviews related work in compiler optimization selection for code size reduction, SBSE, as well as evolutionary algorithms for computationally expensive problems.

A. Compiler Optimization Selection for Code Size Reduction

Two categories of methods are designed to deal with the problem of compiler optimization selection for code size reduction, i.e., evolutionary algorithms and machine learning algorithms.

Cooper *et al.* [4] study the code size reduction of generated object code by using GA. The results of the proposed algorithm are compared with a fixed optimization sequence and optimization sequences selected randomly. It demonstrates that GA can develop a new fixed sequence to reduce code size. Nagiub and Farag [8] add a new genetic operator called pass-over operator in GA to generate good optimization sequences for reducing code size. The added operator enhances the chromosomes' selection for the next generation during iteration. It also proves that the designed approach can assist to optimize the software automatically. Besides the selection of optimization passes with boolean values, other optimization passes with non-Boolean values are also studied. For instance, Chebolu *et al.* [9], [10] apply GA to tune the compiler parameter set. The results obtained by the proposed algorithm show that there is a significant impact of parameter tuning on the code size.

Machine learning algorithms are another major type of approach to select promising compiler optimization passes for code size reduction. These approaches collect the feature information from programs and construct a predictive model based on the training data. The training data contains massive optimization sequences and programs, as well as the performance results of these optimization sequences on each program. Once the model is constructed, given a new program, the feature values are also extracted from the program and then fed into the trained model. Finally, the model can predict the performance of an optimization sequence on the program. However, machine learning models have several drawbacks. First, it requires a relatively large amount of training data, which takes time to obtain the performance results of many optimization sequences on programs. Second, these approaches can only consider limited optimization sequences. Foleiss *et al.* [3] analyze the effect of combinations of compiler optimization passes on code size through the association rule generation algorithm. It is capable of figuring out which combinations of optimization passes bring out similar code size reduction. Their experimental results indicate that the relationship among optimization passes can generate reduced code. In addition, Milepost GCC [7], an open-source machine learning-based compiler, also provides the function to reduce code size. It combines static program features, run-time dynamic features, and optimization passes to train a machine learning model, then the model can automatically learn the best optimization passes for a new program on a given architecture.

In contrast to previous studies, we attempt to apply the surrogate-assisted memetic algorithm in selecting optimization sequences for code size reduction, which extends its application to computationally expensive problems.

B. Search-Based Software Engineering

SBSE refers to a body of work in which search techniques such as GA are applied to resolve complicated software engineering problems [33]–[35]. So far, enormous search techniques can be applied throughout the whole software life cycle, such as testing, maintenance, etc.

Castelein *et al.* [35] devise three search-based approaches to generate the test data for complex SQL queries. The evaluation results demonstrate that 98.6% of all queries can be covered by GA. Abdessalem *et al.* [36] model the vision-based control system testing as a search-based problem. They combine decision tree models with multiobjective search algorithms and generate 78% more pivotal test scenarios. Afterwards, search algorithms are also applied to other testing problems, such as function testing [37], energy testing [38], mutation testing [39], as well as regression testing [40]. Moreover, evolutionary computation has also been applied to other phases of the software life cycle. For example, Soltani *et al.* [41] utilize an interesting guided GA with postfailure analysis to reproduce failure for large real-world programs. An empirical investigation shows that 82% of the crash can be reproduced using this approach. To solve the next release problem better, the authors in [42] and [43] put forward a hybrid ACO algorithm and a multilevel algorithm. Other work about applications of optimization algorithms in software engineering includes coding tools and techniques [44], security and protection [45], etc.

As shown in the above applications in SBSE, we consider the compiler optimization selection for code size reduction as a search task, then try to adopt the surrogate-assisted memetic algorithm to search for a better optimization sequence for programs.

C. Evolutionary Algorithms for Computationally Expensive Problems

When applying evolutionary algorithms in a number of applications, there is no fitness function or the fitness evaluation takes a lot of time. Thus, it will lead to computationally expensive problems.

In general, the approximation in optimization usually utilizes three levels of approaches, i.e., the problem approximation, the functional approximation, and the evolutionary approximation [46].

The problem approximation uses an approximate problem to replace the original problem, which makes it relatively easy to solve. For example, network functions virtualization (NFV) is the basis of distributed cloud networking, the goal of the NFV service distribution problem is to determine the network resources and allocation of the cloud. Feng *et al.* [47] convert the problem into a minimum cost multicommodity-chain network design problem on a cloud-augmented graph, which is proven to be NP-hard.

The functional approximation means that an explicit fitness function is designed in evolutionary computation. Gavalas *et al.* [48] provide several approximation algorithms for the arc orienteering problem in directed graphs and undirected graphs, respectively. A mathematical model can be designed to evaluate the performance of a turbine blade, instead of running the wind tunnel experiments [49].

Besides the aforementioned two kinds of approximations, a great deal of previous research into approximations has focused on evolutionary approximation. These approaches apply approximate models, often known as surrogates, in fitness evaluations to decrease the actual calculation. For example, the individual-based models mainly select some of the individuals to evaluate fitness using an approximate model [50], [51], while others are evaluated by the original fitness function. The generation-based models focus on determining which generations adopt the real fitness function, while other generations are estimated [52], [53]. Nguyen *et al.* [54] develop a new surrogate assisted genetic programming for the automated design of dispatching rules. Their algorithm calculates the best individual of each generation using the fitness function, which has shown great potential for the automated design system. For the optimization of large scale expensive problems, Sun *et al.* [31] consider using the positional relationship between individuals in competitive swarm optimizer to estimate the fitness. An empirical study shows that their approach outperforms the original competitive swarm optimizer. For the high-dimensional expensive problems, Sun *et al.* [55] skillfully combine a surrogate-assisted particle swarm optimization (PSO) algorithm and a surrogate-assisted social learning-based PSO to find high-quality solutions. Furthermore, multiple surrogate models are adopted to solve bilevel optimization problems [56] and adaptive model selection strategies are proposed to choose an appropriate surrogate for replacing the original fitness function [57].

To the best of our knowledge, the application of the surrogate model has so far been scarcely studied to achieve selecting compiler optimization passes for code size reduction. In this study, we present a surrogate-assisted memetic algorithm toward addressing the research gap.

VI. Conclusion

In this study, we propose a novel approach, SMARTEST, to select good optimization sequences for code size reduction. SMARTEST builds on the surrogate model and the local search to replace the original individual with its local optimum by using the approximate fitness score instead of the actual fitness score. Then our approach is evaluated on the compiler GCC and the cBench benchmark suite. The results indicate that SMARTEST can generate smaller binary code and outperform GA and the standard level -Os. Furthermore, by comparing SMARTEST with its variants, we demonstrate that the local search is beneficial for tackling the challenge of neighborhood exploitation.

Additionally, the surrogate model contributes to addressing the challenge of expensive fitness evaluation in selecting optimization sequences.

For the future work, we intend to extend SMARTEST to support multiobjective optimization, rather than only the code size of executable files. In fact, it is necessary to generate executable files that follow specific execution time, resource consumption, and other requirements.

Acknowledgment

The authors would like to thank the anonymous reviewers and editors for their helpful comments.

References

[1] A. Serranocases, Y. Morilla, P. Martinholgado, S. Cuencaasensi, and A. Martinezalvarez, "Nonintrusive automatic compiler-guided reliability improvement of embedded applications under proton irradiation," *IEEE Trans. Nucl. Sci.*, vol. 66, no. 7, pp. 1500–1509, Jul. 2019.

[2] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Comput. Surv.*, vol. 51, no. 5, pp. 1–42, 2018.

[3] J. H. Foleiss, A. F. da Silva, and L. B. Ruiz, "The effect of combining compiler optimizations on code size," in *Proc. 30th Int. Conf. Chilean Comput. Sci. Soc.*, 2011, pp. 187–194.

[4] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *Proc. ACM Sigplan Workshop Lang., Compilers, Tools Embedded Syst.*, 1999, pp. 1–9.

[5] L. G. Martins, R. Nobre, J. M. Cardoso, A. C. Delbem, and E. Marques, "Clustering-based selection for the exploration of compiler optimization sequences," *ACM Trans. Architecture Code Optim.*, vol. 13, no. 1, pp. 1–28, 2016.

[6] N. Narayanamurthy, K. Pattabiraman, and M. Ripeanu, "Finding resilience-friendly compiler optimizations using meta-heuristic search techniques," in *Proc. 12th Eur. Dependable Comput. Conf.*, 2016, pp. 1–12.

[7] G. Fursin *et al.*, "Milepost gcc: Machine learning enabled self-tuning compiler," *Int. J. Parallel Program.*, vol. 39, no. 3, pp. 296–327, 2011.

[8] M. Nagiub and W. Farag, "Automatic selection of compiler options using genetic techniques for embedded software design," in *Proc. 14th Int. Symp. Comput. Intell. Inform.*, 2013, pp. 69–74.

[9] N. B. S. Chebolu, R. Wankar, and R. R. Chillarige, "GA-based compiler parameter set tuning," in *Proc. Artif. Intell. Evol. Algorithms Eng. Syst.*, 2015, pp. 197–203.

[10] N. S. Chebolu, R. Wankar, and R. R. Chillarige, "Tuning the optimization parameter set for code size," in *Proc. Int. Workshop Multi-disciplinary Trends Artif. Intell.*, 2012, pp. 214–223.

[11] G. Fursin, "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. [Online]. Available: http://cTuning.org/cbench

[12] L. D. Whitley, "The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best," *ICGA*, vol. 89, pp. 116–123, 1989.

[13] A. Lipowski and D. Lipowska, "Roulette-wheel selection via stochastic acceptance," *Physica A: Statist. Mech. Appl.*, vol. 391, no. 6, pp. 2193–2196, 2012.

[14] W. M. Spears, "Crossover or mutation?" *Found. Genetic Algorithms*, vol. 2, pp. 221–237, 1993.

[15] K. Deb and R. B. Agrawal, "Simulated binary crossover for continuous search space," *Complex Syst.*, vol. 9, no. 2, pp. 115–148, 1995.

[16] F. Neri and C. Cotta, "Memetic algorithms and memetic computing optimization: A literature review," *Swarm Evol. Comput.*, vol. 2, pp. 1–14, 2012.

[17] A. Liaw and M. Wiener, "Classification and regression by randomforest,*" R News*, vol. 2, no. 3, pp. 18–22, 2002.

[18] L. P. Cáceres, B. Bischl, and T. Stützle, "Evaluating random forest models for irace," in *Proc. Genet. Evol. Comput. Conf. Companion*, 2017, pp. 1146–1153.

[19] H. Wang and Y. Jin, "A random forest-assisted evolutionary algorithm for data-driven constrained multiobjective combinatorial optimization of trauma systems," *IEEE Trans. Cybern.*, vol. 50, no. 2, pp. 536–549, Feb. 2020.

[20] Y. Sun, H. Wang, B. Xue, Y. Jin, G. G. Yen, and M. Zhang, "Surrogate-assisted evolutionary deep learning using an end-to-end random forest-based performance predictor," *IEEE Trans. Evol. Comput.*, vol. 24, no. 2, pp. 350–364, Apr. 2020.

[21] S. K. Dasari, A. Cheddad, and P. Andersson, "Random forest surrogate models to support design space exploration in aerospace use-case," in *Proc. Int. Conf. Artif. Intell. Appl. Innovations*, 2019, pp. 532–544.

[22] G. Fursin, J. Cavazos, M. O'Boyle, and O. Temam, "Midatasets: Creating the conditions for a more realistic evaluation of iterative optimization," in *Proc. Int. Conf. High-Perform. Embedded Architectures Compilers*, 2007, pp. 245–260.

[23] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos, "Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning," *ACM Trans. Architecture Code Optim.*, vol. 14, no. 3, 2017, Art. no. 29.

[24] A. H. Ashouri, A. Bignoli, G. Palermo, and C. Silvano, "Predictive modeling methodology for compiler phase-ordering," in *Proc. 7th Workshop Parallel Program. Run-Time Manage. Techn. Many-core Architectures 5th Workshop Des. Tools Architectures Multicore Embedded Comput. Platforms*, 2016, pp. 7–12.

[25] S. Kulkarni and J. Cavazos, "Mitigating the compiler optimization phase-ordering problem using machine learning," in *Proc. ACM Int. Conf. Object-Oriented Program., Syst., Lang., Appl.*, 2012, pp. 147–162.

[26] F. Li, F. Tang, and Y. Shen, "Feature mining for machine learning based compilation optimization," in *Proc. Int. Conf. Innov. Mobile Internet Services Ubiquitous Comput.*, 2014, pp. 207–214.

[27] M. Boussaa, O. Barais, B. Baudry, and G. Sunyé, "Notice: A framework for non-functional testing of compilers," in *Proc. IEEE Int. Conf. Softw. Quality, Rel. Secur.*, 2016, pp. 335–346.

[28] S. Purini and L. Jain, "Finding good optimization sequences covering program space," *ACM Trans. Architecture Code Optim.*, vol. 9, no. 4, 2013, Art. no. 56.

[29] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *Proc. Int. Symp. Search Based Softw. Eng.*, 2011, pp. 33–47.

[30] H. R. Medeiros, D. M. Izidio, A. P. d. A. Ferreira, and E. N. daS. Barros, "Latin hypercube initialization strategy for design space exploration of deep neural network architectures," in *Proc. Genetic Evol. Comput. Conf. Companion*, 2019, pp. 295–296.

[31] C. Sun, J. Ding, J. Zeng, and Y. Jin, "A fitness approximation assisted competitive swarm optimizer for large scale expensive optimization problems,*" Memetic Comput.*, vol. 10, no. 2, pp. 123–134, 2018.

[32] W. Zhong, C. Qiao, X. Peng, Z. Li, C. Fan, and F. Qian, "Operation optimization of hydrocracking process based on Kriging surrogate model," *Control Eng. Pract.*, vol. 85, pp. 34–40, 2019.

[33] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 1–61, 2012.

[34] A. Arcuri, G. Fraser, and J. P. Galeotti, "Automated unit test generation for classes with environment dependencies," in *Proc. 29th ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2014, pp. 79–90.

[35] J. Castelein, M. Aniche, M. Soltani, A. Panichella, and A. van Deursen, "Search-based test data generation for SQL queries," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 1220–1230.

[36] R. B. Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing vision-based control systems using learnable evolutionary algorithms," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 1016–1026.

[37] J. Ferrer, P. M. Kruse, F. Chicano, and E. Alba, "Search based algorithms for test sequence generation in functional testing," *Inf. Softw. Technol.*, vol. 58, pp. 419–432, 2015.

[38] R. Jabbarvand, J.-W. Lin, and S. Malek, "Search-based energy testing of android," in *Proc. 41st Int. Conf. Softw. Eng.*, 2019, pp. 1119–1130.

[39] E. Omar, S. Ghosh, and D. Whitley, "Constructing subtle higher order mutants for Java and AspectJ programs," in *Proc. IEEE 24th Int. Symp. Softw. Rel. Eng.*, 2013, pp. 340–349.

[40] J. Shelburg, M. Kessentini, and D. R. Tauritz, "Regression testing for model transformations: A multi-objective approach," in *Proc. Int. Symp. Search Based Softw. Eng.*, 2013, pp. 209–223.

[41] M. Soltani, A. Panichella, and A. Van Deursen, "A guided genetic algorithm for automated crash reproduction," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 209–220.

[42] H. Jiang, J. Zhang, J. Xuan, Z. Ren, and Y. Hu, "A hybrid ACO algorithm for the next release problem," in *Proc. Int. Conf. Softw. Eng. Data Mining*, 2010, pp. 166–171.

[43] J. Xuan, H. Jiang, Z. Ren, and Z. Luo, "Solving the large scale next release problem with a backbone-based multilevel algorithm," *IEEE Trans. Softw. Eng.*, vol. 38, no. 5, pp. 1195–1212, Sep./Oct. 2012.

[44] J. Kukunas, R. D. Cupper, and G. M. Kapfhammer, “A genetic algorithm to improve linux kernel performance on resource-constrained devices,” in *Proc. 12th Annu. Conf. Companion Genetic Evol. Comput.*, 2010, pp. 2095–2096.

[45] G. Dozier, D. Brown, H. Hou, and J. Hurley, “Vulnerability analysis of immunity-based intrusion detection systems using genetic and evolutionary hackers,” *Appl. Soft Comput.*, vol. 7, no. 2, pp. 547–553, 2007.

[46] Y. Jin, “A comprehensive survey of fitness approximation in evolutionary computation,*” Soft Comput.*, vol. 9, no. 1, pp. 3–12, 2005.

[47] H. Feng, J. Llorca, A. M. Tulino, D. Raz, and A. F. Molisch, “Approximation algorithms for the NFV service distribution problem,” in *Proc. IEEE Conf. Comput. Commun.*, 2017, pp. 1–9.

[48] D. Gavalas, C. Konstantopoulos, K. Mastakas, G. Pantziou, and N. Vathis, “Approximation algorithms for the arc orienteering problem,” *Inf. Process. Lett.*, vol. 115, no. 2, pp. 313–315, 2015.

[49] D. John, *Computational Fluid Dynamics: The Basics With Applications*. New York, NY, USA: McGraw Hill, 1995.

[50] J. Branke and C. Schmidt, “Faster convergence by means of fitness estimation,” *Soft Comput.*, vol. 9, no. 1, pp. 13–20, 2005.

[51] Y. Jin, M. Olhofer, and B. Sendhoff, “A framework for evolutionary optimization with approximate fitness functions,” *IEEE Trans. Evol. Comput.*, vol. 6, no. 5, pp. 481–494, Oct. 2002.

[52] I. Loshchilov, M. Schoenauer, and M. Sebag, “Self-adaptive surrogate-assisted covariance matrix adaptation evolution strategy,” in *Proc. 14th Annu. Conf. Genetic Evol. Comput.*, 2012, pp. 321–328.

[53] H. Yu, Y. Tan, C. Sun, and J. Zeng, “A generation-based optimal restart strategy for surrogate-assisted social learning particle swarm optimization,” *Knowl.-Based Syst.*, vol. 163, pp. 14–25, 2019.

[54] S. Nguyen, M. Zhang, and K. C. Tan, “Surrogate-assisted genetic programming with simplified models for automated design of dispatching rules,” *IEEE Trans. Cybern.*, vol. 47, no. 9, pp. 2951–2965, Sep. 2017.

[55] C. Sun, Y. Jin, R. Cheng, J. Ding, and J. Zeng, “Surrogate-assisted cooperative swarm optimization of high-dimensional expensive problems,” *IEEE Trans. Evol. Comput.*, vol. 21, no. 4, pp. 644–660, Aug. 2017.

[56] M. M. Islam, H. K. Singh, and T. Ray, “A surrogate assisted approach for single-objective bilevel optimization,” *IEEE Trans. Evol. Comput.*, vol. 21, no. 5, pp. 681–696, Oct. 2017.

[57] H. Yu, Y. Tan, C. Sun, J. Zeng, and Y. Jin, “An adaptive model selection strategy for surrogate-assisted particle swarm optimization algorithm,” in *Proc. IEEE Symp. Ser. Comput. Intell.*, 2016, pp. 1–8.