2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) | 978-1-6654-0296-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICSE43902.2021.00110

Efficient Compiler Autotuning via Bayesian Optimization

Junjie Chen[†]
College of Intelligence and Computing
Tianjin University
Tianjin, China
junjiechen@tju.edu.cn

Ningxin Xu
College of Intelligence and Computing
Tianjin University
Tianjin, China
xnxdw@tju.edu.cn

Peiqi Chen
College of Intelligence and Computing
Tianjin University
Tianjin, China
chenpeiqi@tju.edu.cn

Hongyu Zhang
The University of Newcastle
Callaghan, NSW, Australia
hongyu.zhang@newcastle.edu.au

***Abstract*—A typical compiler such as GCC supports hundreds of optimizations controlled by compilation flags for improving the runtime performance of the compiled program. Due to the large number of compilation flags and the exponential number of flag combinations, it is impossible for compiler users to manually tune these optimization flags in order to achieve the required runtime performance of the compiled programs. Over the years, many compiler autotuning approaches have been proposed to automatically tune optimization flags, but they still suffer from the efficiency problem due to the huge search space. In this paper, we propose the first Bayesian optimization based approach, called BOCA, for efficient compiler autotuning. In BOCA, we leverage a tree-based model for approximating the objective function in order to make Bayesian optimization scalable to a large number of optimization flags. Moreover, we design a novel searching strategy to improve the efficiency of Bayesian optimization by incorporating the impact of each optimization flag measured by the tree-based model and a decay function to strike a balance between exploitation and exploration. We conduct extensive experiments to investigate the effectiveness of BOCA on two most popular C compilers (i.e., GCC and LLVM) and two widely-used C benchmarks (i.e., cBench and PolyBench). The results show that BOCA significantly outperforms the state-of-the-art compiler autotuning approaches and Bayesion optimization methods in terms of the time spent on achieving specified speedups, demonstrating the effectiveness of BOCA.**

***Index Terms*—Compiler Autotuning, Bayesian Optimization, Compiler Optimization, Configuration**

I. Introduction

Compilers (e.g., GCC [1] and LLVM [2]) are responsible for transforming a source program written in a certain programming language (e.g., C and C++) into an executable program [3]. To improve the runtime performance of compiled programs [4], a compiler supports a lot of code optimizations, each of which can be enabled or disabled by a compilation flag. For example, GCC has hundreds of compilation flags to support optimizations. One flag example is `-finline-small-functions`, which can enable the optimization that integrates functions into their callers when their bodies are smaller than the expected function call code. However, since the effect of compiler optimizations depends on the program characteristics (e.g., program structures), the same optimizations do not always lead to the same improvement in runtime performance when they are applied to different programs [5]. Moreover, due to the large number of optimization flags, there are an exponential number of combinations of flags. It is challenging for users to understand all the flags and their combinations and properly determine which flags should be enabled or disabled in order for the compiled programs to achieve the required runtime performance [6].

To ease usage of compiler optimizations, compiler developers pre-define several optimization levels (e.g., `-O1`, `-O2`, `-O3` in GCC and LLVM), each of which enables a fixed set of optimization flags and aims to achieve a certain optimization goal in general. That is, users can specify an optimization level to impose a default set of optimization flags for the compiler. However, due to the large differences of program characteristics, there is no optimization level that can guarantee to achieve the required runtime performance (measured in terms of the execution time of the compiled program) for every program. Therefore, it is necessary for each specific program to carefully tune the optimization flags in order to achieve required runtime performance, instead of directly using the pre-defined optimization levels. This is especially essential for the applications that are sensitive to runtime performance.

In the literature, many compiler autotuning approaches have been proposed to automatically tune optimization flags in order to achieve required runtime performance for a given program [6]–[9]. In general, they iteratively test various settings of optimization flags (we call an optimization flag setting *an optimization sequence* following the existing work [6]) with certain search strategies (e.g., random search [5] or genetic algorithm [10]) and then output the best one when reaching the

[†] Junjie Chen is the corresponding author.

978-1-6654-0296-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSE43902.2021.00110

terminating condition. Although these approaches have been demonstrated to be effective to some degree, they still suffer from the efficiency problem [5], [10], [11]. More specifically, with tens to hundreds of compiler optimization flags provided by a compiler, the number of flag combinations is exponential, leading to a very huge search space. Therefore, the existing approaches have to compile and execute programs with a large number of optimization sequences during the search process, which incurs large cost especially when the compilation and execution are lengthy. As an example, for the program `consumer_jpeg_c` with GCC used in our study (presented in Section IV), the state-of-the-art approach Irace [11] cannot achieve the required runtime performance within the given time period (i.e., 3.5 hours). Therefore, it is very important to improve the efficiency of compiler autotuning.

To solve the problem, in this paper we propose the first **B**ayesian **O**ptimization based approach for **C**ompiler **A**utotuning, called **BOCA**. Bayesian optimization is recognized as an effective method to optimize an expensive-to-evaluate objective function [12]. It uses the accumulated knowledge in the known area of the search space to guide samplings in the remaining area in an iterative process so as to find the optimal sample efficiently. Over the years, Bayesian optimization has been applied to a wide range of applications such as hyperparameter optimization in deep learning [13]. However, it is hard for traditional Bayesian optimization methods to scale to high-dimensional data due to their inherent mechanisms (e.g., relying on Gaussian Process [14] to approximate the objective function) [15]. Thus, they are inefficient for compiler autotuning, where a large number of optimization flags need to be tuned. Some advanced Bayesian optimization methods have been proposed to handle high-dimensional data [16], [17]. However, since they are not designed for compiler autotuning, they do not take the characteristics of compiler autotuning into consideration (e.g., only a small number of optimization flags, referred to as *impactful optimizations*, can have noticeable impact on the runtime performance of a specific program). As a result, the direct application of the existing Bayesian optimization methods in compiler autotuning is not efficient (which will be demonstrated in our study presented in Section V-B).

In this paper, we carefully design BOCA to improve the efficiency of compiler autotuning. To make BOCA scalable to a large number of optimization flags, we incorporate Random Forest (a tree-based ensemble learning algorithm) [18] to build a probabilistic model to approximate the objective function based on the already evaluated samples. Then, BOCA uses the model to predict the quality of unevaluated optimization sequences for the next sampling. Due to the sheer size of optimization sequences, it is very costly to predict all of them in each iteration. Therefore, we design a selection strategy in BOCA to select only a subset of candidate optimization sequences so that an optimization sequence that can achieve required runtime performance can be efficiently found by just predicting the subset. In particular, our selection strategy balances between exploitation and exploration:

- *Exploitation*: By taking the advantage of flag importance measured based on the tree-based model, BOCA identifies the impactful optimizations and then fully enumerates their combinations;
- *Exploration*: BOCA explores a number of settings of the remaining optimizations (called less-impactful optimizations) with a decay function, which is helpful to avoid local optimum (by exploring less-impactful optimizations) and further improve the efficiency (by utilizing a decay function).

We conducted extensive experiments to evaluate the effectiveness of BOCA on two most widely-used C compilers (i.e., GCC [1] and LLVM [2]) [19]–[22] and on two public C benchmarks (i.e., cBench [23] and PolyBench [24]) widely used in the existing compiler autotuning work [25]–[28]. Our experimental results demonstrate that BOCA does spend less time to achieve the required runtime performance and significantly outperforms the existing compiler autotuning approaches (i.e., one baseline RIO [5] and two state-of-the-art approaches GA [10] and Irace [11]). The average improvement (in terms of the time spent on achieving the required runtime performance) is 42.30%$\sim$78.04%. Besides, we compared BOCA with the existing Bayesian optimization methods (i.e., one traditional method ε-PAL [29] and two advanced methods FLASH [30] and TPE [16]). The results show that BOCA significantly outperforms all of them, confirming the necessity of designing a novel Bayesian optimization method for compiler autotuning. Furthermore, our evaluation confirms the effectiveness of the selection strategy and the decay function used in BOCA.

Our work makes the following major contributions:

- We propose BOCA, the first Bayesian optimization based approach for compiler autotuning, which significantly improves the efficiency of compiler autotuning.
- We design a novel selection strategy in BOCA to select a subset of candidate optimization sequences by exploiting impactful optimizations with a tree-based model and exploring less-impactful optimizations with a decay function.
- We conducted extensive experiments on two widely-used C compilers (i.e., GCC and LLVM) and two widely-used C benchmarks (i.e., cBench and PolyBench). The results demonstrate the effectiveness of BOCA.

II. Background and Related Work

A. Compiler Autotuning

In the literature, a large amount of research work focuses on compiler autotuning [6]–[9]. Here, we briefly introduce some typical and state-of-the-art approaches, which are also used for comparison in our study (presented in Section IV).

- **Random Iterative Optimization (RIO)** [5]: In each iteration, it randomly sets each optimization flag (enabled/disabled), and then evaluates the performance of the compiled program under the optimization sequence. This process is repeated until the terminating condition is reached. RIO has been demonstrated to be effective and is regarded as the baseline in our study.

- **Genetic Algorithm based Iterative Optimization (GA)** [10]: GA first constructs an initial set of chromosomes, each of which is a random setting of optimization flags. In each iteration, new chromosomes are produced via crossover and mutation operations where the former exchanges the settings of some optimization flags in two chromosomes to produce new chromosomes and the latter randomly flips the setting of an optimization flag in a chromosome to produce a new one. Next, new chromosomes are evaluated and the set of chromosomes are updated based on the evaluation results. This process is repeated until the terminating condition is reached. GA is a state-of-the-art approach.
- **Irace-based Iterative Optimization (Irace)** [11]: Irace learns the sampling distribution for each optimization flag, which is used to set each optimization flag, during the iterative process. It first constructs a set of random settings of optimization flags as the initial set, and learns the initial sampling distribution for each optimization flag. In each iteration, it produces new settings of optimization flags according to the learned sampling distributions, and then new optimization flag settings are evaluated and the sampling distribution for each optimization flag is updated accordingly. This process is repeated until the terminating condition is reached. Irace is also a state-of-the-art approach.

Besides, there are some supervised approaches based on offline learning for compiler autotuning [6], [7]. Here, we did not compare with them since their effectiveness heavily depends on the quality of training data used for offline learning and collecting a large amount of training data (an instance includes a program, a setting of optimization flags, and the achieved performance of the compiled program under the setting) is very costly in practice. Although the above-introduced search-based approaches have been demonstrated to be effective and do not rely on a large amount of training data, they still suffer from the efficiency issue as presented in Section I. Therefore, improving the efficiency of compiler autotuning is very important. That is, it is necessary to propose a novel approach that can spend less time to find an optimization sequence that can achieve required runtime performance.

B. Bayesian Optimization

In this paper, we incorporate Bayesian optimization to improve the efficiency of compiler autotuning. Bayesian optimization is a method to optimize an objective function $f(.)$, which is expensive to evaluate [12]. Its core idea is to use the accumulated knowledge in the known area of the search space to guide samplings in the remaining area in order to find the optimal sample more efficiently. It is an on-the-fly iterative process consisting of two main steps: 1) building a surrogate model for approximating the objective function based on already measured samples, and 2) guiding the further sampling based on the surrogate model and an acquisition function. In traditional Bayesian optimization, Gaussian Process (GP) is used to build the surrogate model [14]. An acquisition function is used to decide where to sample next so that an improvement over the current best observation is likely to be achieved. The widely-used acquisition functions include Expected Improvement, Maximum Variance, and Maximum Mean [30], [31]. According to the acquisition function, the sample with the best value will be selected and measured. Then, the sample with its measured observation will be used to update the surrogate model for the next iteration. For example, **ε-PAL** [29] is recognized as a typical Bayesian optimization method, which uses Maximum Variance as the acquisition function. However, traditional Bayesian Optimization methods cannot scale to high-dimensional data (e.g., the large number of optimization flags) [30], and thus they have not been applied to the area of compiler autotuning before.

In a similar area, i.e., configurable software systems, a state-of-the-art approach, called **FLASH** [30], is based on Bayesian optimization. To overcome the shortcoming of GP, FLASH replaces GP with CART (Classification and Regression Trees) [32], and uses Maximum Mean as the acquisition function since CART outputs only one value. FLASH can handle more high-dimensional data, but it is still costly even unaffordable for compiler autotuning. This is because the number of flags is large (e.g., the number of tuned flags for GCC in our study is 71), leading to an extremely large number of flag combinations, and FLASH has to enumerate and predict all unevaluated optimization sequences in each iteration. Furthermore, compilers are also highly configurable systems. In the area of configuring software systems, many approaches have been proposed to *predict performance of a configurable software system* by training a model using a sample of configurations [33]–[39]. Different from them, our work aims to find an optimization sequence, under which *the compiled program can achieve required runtime performance*.

In this paper, we design a novel Bayesian optimization based approach for *efficient* compiler autotuning, which addresses the limitations of existing Bayesian optimization methods.

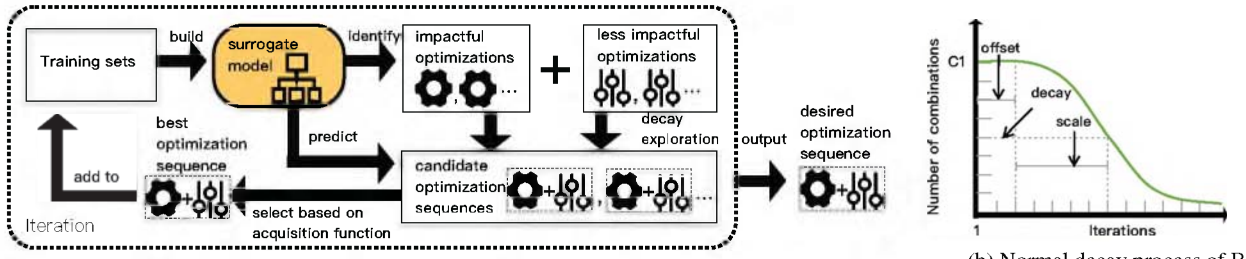
III. Approach

A. Problem Definition

We denote the set of optimization flags supported by a compiler as $O = \{o_1, o_2, \ldots, o_m\}$, where m is the size of the optimization set and o_i can be set to 0 or 1 (0/1 refers to disabling/enabling it). A specific setting of optimization flags is called an *optimization sequence* [6]. All the possible optimization sequences form the whole optimization space, which is denoted as S. Since the setting of each flag has two optional values (i.e., 0 and 1) and the number of optimization flags is m, the size of the optimization space is $|S| = 2^m$. In this paper, we aim to *improve the efficiency of compiler autotuning*, i.e., spending less time to find an optimization sequence in S that can achieve the required runtime performance for a given program. We call such an optimization sequence *desired optimization sequence*.

B. Approach Overview

Due to the huge optimization space and large evaluation cost, it is challenging to efficiently find a desired optimiza-



(a) Overview of BOCA

(b) Normal decay process of BOCA

Fig. 1: An illustration of BOCA

tion sequence. In this paper, we propose the first **B**ayesian **O**ptimization based approach for **C**ompiler **A**utotuning, named **BOCA**, since Bayesian optimization is well recognized as an efficient method to optimize an expensive-to-evaluate objective function [12]. However, as presented in Section II-B, due to the high-dimensional data (i.e., the large number of optimization flags) in compiler autotuning, both traditional Bayesian optimization (e.g., ε-PAL) and the state-of-the-art Bayesian optimization (e.g., FLASH) methods used for configuring software systems are not affordable. Thus, we carefully design the search process in our approach by incorporating Random Forest (RF) [18] and proposing a novel selection strategy for candidate optimization sequences in order to overcome the thorny efficiency issue. The selection strategy aims to select a subset of candidate optimization sequences, which is very likely to include a desired one, so that a desired optimization sequence can be found more efficiently by just predicting the subset instead of predicting all unevaluated optimization sequences. In particular, RF is a state-of-the-art machine learning algorithm, which is able to effectively learn the interactions and dependencies between optimization flags.

Figure 1a shows the overview of BOCA. In each iteration, BOCA first learns a surrogate model via Random Forest based on the training set. It then selects a set of candidate optimization sequences to be predicted by the surrogate model. Finally, it selects the best optimization sequence among these candidates according to a acquisition function, which will be evaluated and then used to update the surrogate model. In the following, we introduce our surrogate model and acquisition function in Section III-C, and present the selection strategy for candidate optimization sequences in Section III-D and the overall algorithm of BOCA in Section III-E.

C. Building a Surrogate Model

There are two requirements for building a surrogate model: a training set and a model-building algorithm. In BOCA, a training instance is an optimization sequence and the execution time of a given program after compilation under the optimization sequence. That is, the vector of an optimization sequence is the feature vector of the training instance, and the execution time is its label. There are two sources of training instances. Initially, BOCA randomly selects n optimization sequences and evaluates them to get the corresponding execution time, which are used as the initial training set. In each subsequent iteration, BOCA selects the best optimization sequence from the remaining area of optimization space according to the acquisition function. It then adds the best one and the associated program execution time to the training set.

Based on a training set, BOCA adopts Random Forest (RF) to build a surrogate model, instead of GP in traditional Bayesian optimization. RF is an ensemble machine learning technique that merges multiple decision trees together to get a more accurate and stable prediction, where each decision tree is built based on a subset of training data randomly sampled from the training set and a subset of features randomly sampled from the whole feature set [18]. The reasons why we replace GP with RF are twofold. First, GP cannot scale to high dimensional data [40]. In particular, it has been found that recent work using GP-based Bayesian optimization in the SE community is limited to around 10 dimensional data (i.e., 10 configuration options) [30], [41], which is much smaller than the number of optimization flags. However, RF performs well for high-dimensional data [42], [43]. Second, GP is very sensitive to its parameters [30], but RF performs stably because of its ensemble strategy. In particular, through such a state-of-the-art machine learning algorithm, it can more effectively learn the interactions and dependencies between optimization flags to guarantee the effectiveness of BOCA.

Acquisition Function. BOCA uses Expected Improvement (EI) [31] as the acquisition function since it considers both exploration (measured by the standard deviation of a prediction) and exploitation (measured by the mean of a prediction) when determining the next optimization sequence to be measured. In GP, the mean and the standard deviation of a prediction can be directly outputted. BOCA uses RF to merge multiple decision trees, and thus it acquires the mean and the standard deviation of a prediction based on the prediction of each decision tree. More specifically, given a candidate optimization sequence, each decision tree makes a prediction for it and then BOCA calculates the mean and the standard deviation based on all these predictions made by decision trees.

D. Selecting Candidate Optimization Sequences

Based on the knowledge learnt from the optimization sequences that have already been measured, BOCA searches for an desired optimization sequence in the remaining space.

Due to the huge optimization space, it is extremely costly to predict all the remaining optimization sequences using the surrogate model in order to find a desired optimization sequence. Therefore, to reduce the cost, BOCA selects a subset of candidate optimization sequences that is very likely to include a desired one, so that the desired one can be efficiently found by just predicting this subset. However, selecting such a subset of candidate optimization sequences is also challenging since the optimization space is extremely huge.

To address this challenge, we carefully design a selection strategy. The key insight is that for a given program, only a small number of optimizations can have a large influence on its execution time (called *impactful optimizations*). Thus, sufficiently exploiting the impactful optimizations is more likely to find a desired solution. However, precisely identifying those impactful optimizations is difficult. To reduce the influence of noise, it is also necessary to explore other optimizations (called *less-impactful optimizations*) and the degree of exploration should be decayed with the precision of identifying impactful optimizations improving, so as to avoid incurring overmuch cost. By considering both exploitation and exploration for the optimization space, BOCA selects a subset of candidate optimization sequences that is very likely to include a desired one. In the following, we present the processes of *impactful optimization identification* (Section III-D1) and *less-impactful optimization exploration with decay* (Section III-D2) in detail.

1) Impactful Optimization Identification.: BOCA adopts *Gini importance* [44] to measure the impact of optimizations since BOCA depends on multiple decision trees and Gini importance is often used by tree-based machine learning techniques to conduct feature selection. In a decision tree, each node has Gini importance that measures the total decrease of Gini impurity in the node after splitting using a feature (referring to an optimization flag in our work). Here, Gini impurity in a node is a measure of the likelihood of a randomly chosen optimization sequence from the set of optimization sequences in the node would be incorrectly labeled if it is randomly labeled according to the label distribution of the set in the node. Please note that in our problem, each feature actually has two optional values, i.e., 0 and 1, and thus a feature can be used for splitting at most one node in a decision tree. Therefore, Gini importance of a node is equal to Gini importance of the feature used for splitting on this node in a decision tree. More specifically, the calculation of Gini importance for a node d is shown in Formula 1.

$$G(d) = [I(d) - \omega_{left} * I(left) - \omega_{right} * I(right)] * \frac{n_d}{N} \quad (1)$$

where $I(.)$ is Gini impurity of a node, $\omega_{left}/\omega_{right}$ refers to the proportion of optimization sequences reaching the left/right node from a node d, n_d is the number of optimization sequences in the node, and N is the total number of optimization sequences in the whole training set. Further, the calculation of Gini impurity for a node d is shown in Formula 2.

$$I(d) = 1 - \sum_{i=1}^{c} p_i^2 \quad (2)$$

where c is the number of classes in the set and p_i is the probability of picking an optimization sequence with class i from the set. Please note that, although BOCA aims to construct a regression model through RF, each decision tree actually splits the whole range into several intervals as classes during training, enabling the calculation of Gini impurity.

After acquiring Gini importance of each optimization flag in each decision tree (if the feature is used for splitting in the tree), BOCA further calculates the impact of each optimization flag by merging all these decision trees, as shown in Formula 3. In this formula, u is the number of decision trees that use o for splitting on a node, t is the total number of decision trees, and d_i is the node using o for splitting in the i^{th} decision tree.

$$Impact(o) = \frac{\sum_{i=1}^{u} G(d_i)}{t} \quad (3)$$

By prioritizing optimization flags in the descending order of their impact, BOCA identifies top-K optimizations as impactful optimizations. Since impactful optimizations could have large influence on the execution time of the given program, BOCA conducts extensive exploitation of them. That is, BOCA exploits all the combinations of these impactful optimizations. Here, to avoid incurring large costs, K should be small. We discuss the setting of K in our study (Section IV-C) and the impact of K on the results in Section VI-A.

2) Less-impactful Optimization Exploration with Decay: Besides setting these impactful optimizations, a complete optimization sequence also needs to set the remaining optimizations (less-impactful optimizations). Intuitively, for a fixed setting of impactful optimizations, it is acceptable to randomly set the remaining optimizations to form a complete optimization sequence since the remaining optimizations have a slight influence on the execution time of the compiled program. However, it is difficult to precisely determine the impact of each optimization using any machine learning technique, especially when the training set is not large at the first few iterations. That is, the remaining optimizations may also contain really impactful optimizations. To reduce the influence of noise, BOCA further explores many combinations of the less-impactful optimizations for a fixed setting of impactful optimizations. Since it is impossible to explore all the combinations, BOCA randomly selects a certain number of combinations to explore. Another benefit of the random exploration is to avoid local optima. To further reduce the costs incurred by exploration, BOCA gradually decays the degree of exploration (i.e., the number of randomly selected combinations of less-impactful optimizations) when the size of the training set is increasing. This is because as the size of the training set increases, the identification of impactful optimizations could become more precise, and thus it is not necessary to extensively explore the remaining optimizations.

BOCA uses a *exploration decay function* to determine the number of explored combinations of less-impactful optimizations for each fixed setting of impactful optimizations at each iteration. Since in the first few iterations the training-set size is relatively small, it is more likely to have much noise, and thus the decay could be slow in the beginning. Subsequently, the training-set size becomes larger and larger, the noise could be reduced, and thus the decay could be fast. With this intuition, BOCA uses a *normal decay function* as shown in Formula 4.

$$C(i) = C_1 * exp\left(-\frac{max(0, i - \mathit{offset})^2}{2\sigma^2}\right),\ \sigma^2 = -\frac{scale^2}{2\log(decay)} \quad (4)$$

where C_1 is the initial number of explored combinations of less-impactful optimizations for each setting of impactful optimizations, i is the iteration ($i \geq 2$), $C(i)$ is the number of explored combinations of less-impactful optimizations at the i^{th} iteration, and *offset*, *scale*, and *decay* control the decay shape of Gaussian decay function. Figure 1b depicts the decay process of BOCA and the meanings of *offset*, *scale*, and *decay*.

Based on the calculated number of explored combinations, BOCA randomly forms $C(i)$ combinations of less-impactful optimizations for each setting of impactful optimizations, thus the total number of selected candidate optimization sequences is $C(i) * 2^K$ at the i^{th} iteration. By calculating the EI values of these candidate optimization sequences via the surrogate model, BOCA selects the optimization sequence with the best EI value as the optimal one among the set, then measures the execution time of the given program after being compiled under it, and adds it to the training set for the next iteration.

E. Overall Algorithm of BOCA

We formally present BOCA in Algorithm 1. For a given program to be compiled, Algorithm 1 outputs a desired optimization sequence s_{best} that makes the program achieve shorter execution time after being compiled under s_{best}, and also the corresponding execution time $f(s_{best})$. We use $f(s)$ to represent the execution time of a program after being compiled under an optimization sequence s. Lines 1-7 construct the initial training set (denoted as *train*) by randomly generating $size_{initial}$ optimization sequences and measuring their execution time. Lines 8-31 conduct the iteration process of BOCA. Line 9 builds a surrogate model via RF based on the training set. Lines 10-14 acquire $\mathcal{K}$ impactful optimizations by calculating the impact of each optimization and also treat the remaining optimizations as less-impactful optimizations. Line 15 enumerates all the settings of impactful optimizations. Besides setting impactful optimizations, a complete optimization sequence also requires a setting of less-impactful optimizations. Then, Lines 16-21 randomly explore C non-repetitive settings of less-impactful optimizations for each setting of impactful optimizations, where C is calculated based on the normal decay function. In this way, a set of candidate optimization sequences (denoted as *allCandidates*) are obtained by considering both exploitation and exploration. Lines 22-25 use the surrogate model to predict each candidate optimization sequence so as to obtain the mean and standard deviation of each prediction, which are used to calculate the EI value of each candidate. Lines 26-27 obtain the optimization sequence that has the largest EI value among all the candidates and is not in the training set, measure this optimization sequence, and then add it and its execution time to the training set for the next iteration. Lines 28-30 obtain the currently optimal optimization sequence among all the measured ones and Line 32 outputs the finally desired one when the terminating condition is reached.

Algorithm 1: Pseudo-code of BOCA

```
Input : $\mathcal{O}$: a list of optimizations $[o_i | i \in 1 \ldots m]$
        $\mathcal{K}$: the number of impactful optimizations
        $size_{initial}$: the size of the initial training set
Output: $(s_{best}, f(s_{best}))$: (a desired optimization sequence, the execution time
        of the given program after compilation under $s_{best}$)
1  train ← []/* training set to build a surrogate model */
2  foreach $i$ from 1 to $size_{initial}$ do
3      foreach $j$ from 1 to $m$ do
4          $s[j]$ ← random(0, 1)/* randomly set 0 or 1 for $o_j$ in
               an optimization sequence $s$ */
5      end
6      train.add($s$, $f(s)$)
7  end
8  foreach $i$ from 1 to iterations do
9      model ← RandomForest(train)
10     foreach $j$ from 1 to $m$ do
11         importance[$j$] ← getImportance($o_j$, model)/* get the
               impact of $o_j$ */
12     end
13     importantOpts ← getImportantOpts(importance, $\mathcal{O}$, $\mathcal{K}$)/* get
           top-$K$ impactful optimizations */
14     unimportantOpts ← set($\mathcal{O}$) – importantOpts
15     importantSettings ← getAllSettings(importantOpts)/* get all the
           settings of impactful optimizations */
16     allCandidates ← []
17     foreach $j$ from 1 to size(importantSettings) do
18         $C$ ← normalDecay($i$)/* get the number of explored
               settings of less-impactful optimizations */
19         unimportantSettings ← getRandomSettings(unimportantOpts, $C$)
               /* randomly get $C$ non-repetitive settings
               of less-impactful optimizations */
20         allCandidates.add(importantSettings[$j$], unimportantSettings)
               /* get $C$ candidate optimization sequences
               by combining importantSettings[$k$] and each
               in unimportantSettings */
21     end
22     foreach $j$ from 1 to size(allCandidates) do
23         (mean, std) ← model.predict(allCandidates[$j$])
24         ei[$j$] ← EI(mean, std)
25     end
26     bestCandidate ← getBestCandidate(allCandidates, ei) /* get the
           candidate with largest $ei$ and not in $train$ */
27     train.add(bestCandidate, $f$(bestCandidate))
28     if $f$(bestCandidate) < $f(s_{best})$ then
29         $(s_{best}, f(s_{best}))$ ← (bestCandidate, $f$(bestCandidate))
30     end
31 end
32 return $(s_{best}, f(s_{best}))$
```

IV. Evaluation

In the study, we address the following research questions:

- **RQ1**: How does BOCA perform compared with existing compiler autotuning approaches?
- **RQ2**: Does BOCA outperform the existing Bayesian optimization methods in compiler autotuning?
- **RQ3**: Is the selection strategy proposed in BOCA effective?

A. Compilers and Programs

We used two most popular open-source C compilers (GCC [1] and LLVM [2]) [45]–[47] as subjects, and two widely-used C benchmarks (cBench [23] and PolyBench [24]) as programs following the existing compiler autotuning work [5], [25]–[28]. Following the prerequisites of the benchmarks and existing work [5], [48], we used GCC 4.7.7 and LLVM 2.9 for the x86-64 Linux platform. In our study, we tuned 71 optimization flags for GCC and 64 optimization

TABLE I: Basic information of subject programs

ID	Program	#SLOC	Description
C1	consumer_jpeg_c	26,950	Image compression and decompression.
C2	security_sha	297	A secure hash algorithm.
C3	automotive_bitcount	954	Testing bit manipulation abilities.
C4	automotive_susan_e	2,129	Image recognition for edges.
C5	automotive_susan_c	2,129	Image recognition for corners.
C6	automotive_susan_s	2,129	Image smoothing.
C7	bzip2e	7,200	File compression and decompression.
C8	consumer_tiff2rgba	22,321	RGB formatted TIFF image conversion.
C9	telecom_adpcm_c	389	Pulse Code Modulation.
C10	office_rsynth	5,412	Text to speech synthesis program
P1	2mm	252	2 Matrix multiplications.
P2	3mm	267	3 Matrix multiplications.
P3	cholesky	212	Cholesky decomposition.
P4	jacobi-2d	200	2-D Jacobi stencil computation.
P5	lu	210	LU decomposition.
P6	correlation	248	Correlation computation.
P7	nussinov	569	DP for sequence alignment.
P8	symm	231	Symmetric matrix-multiply.
P9	heat-3d	211	Heat equation over 3D data domain.
P10	covariance	218	Covariance computation.

flags for LLVM, including both the optimization flags in `-O3` (the highest optimization level in both GCC and LLVM [26]) and the optimization flags that have been demonstrated to have large influences on runtime performance of compiled programs [5].

Regarding the used benchmarks, we used 20 programs (including 10 programs from cBench and 10 programs from PolyBench) in total. Here, we did not use all the programs from the two benchmarks, since the execution time of the remaining programs cannot be noticeably affected by compiler optimizations although tuning, which has been demonstrated by the existing work [5]. Each program is equipped with an input set, which can be executed to measure the runtime performance of the compiled program. Table I shows the basic information of the used programs in our study, where each column presents the ID ("C" is cBench and "P" is PolyBench), the program name, the number of source lines of code (SLOC), and the brief description about the program, respectively.

The code of BOCA, the experimental data, and the complete list of optimization flags used in our study are available at the project webpage: https://github.com/BOCA313/BOCA.

B. Compared Approaches

We considered three categories of compared approaches.

1) Existing Compiler Autotuning Approaches: As presented in Section II-A, we considered three existing compiler autotuning approaches for comparison, i.e., **RIO**, **GA**, and **Irace**, where RIO is regarded as the baseline while GA and Irace are the state-of-the-art approaches for compiler autotuning.

2) Existing Bayesian Optimization Methods: BOCA is the first Bayesian optimization based approach for compiler autotuning, which incorporates a machine learning technique (RF) and a novel selection strategy for selecting candidate optimization sequences. In the literature [16], [29], [49], [50], many Bayesian optimization methods have been proposed, thus it is interesting to investigate whether or not our proposed Bayesian optimization based approach BOCA can outperform the direct application of existing Bayesian optimization methods for compiler autotuning. Here, we chose the following Bayesian optimization methods for comparison: a traditional method (**ε-PAL**), a state-of-the-art method originally proposed for configuring software systems (**FLASH**), and an advanced general Bayesian optimization method (**TPE** [16]).

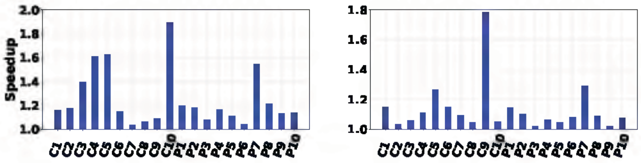
ε-PAL and FLASH have been presented in Section II-B. Here, we introduce TPE briefly. Different from other Bayesian optimization methods, the predictive model in TPE does not predict the posterior probability distribution value for $f(x)$ at a candidate sample x. Instead, it models both the distribution of x given $f(x)$ and the distribution of $f(x)$, and then derives the posterior probability distribution value for $f(x)$ at x when calculating the acquisition function value of x.

3) Selection Strategy: An important component in BOCA for improving efficiency is the selection strategy for candidate optimization sequences, thus it is important to investigate its contribution to BOCA. An advanced strategy to construct a set of candidate optimization sequences in Bayesian optimization is the local search strategy proposed in SMAC [17], which conducts local search based on n_b best optimization sequences measured by the acquisition function. More specifically, for each best optimization sequence, it first constructs m optimization sequences by flipping one flag setting where m is the number of optimization flags, and then uses the optimization sequence with the best acquisition function value as a candidate optimization sequence produced from the best optimization sequence. Also, this strategy randomly constructs n_r candidate optimization settings. That is, the total number of candidate optimization settings constructed via this strategy is $n_b + n_r$. Therefore, we compared BOCA with the variant that replaces the selection strategy used in BOCA with the local search strategy used in SMAC. We call this variant **BOCA$_s$**.

C. Implementations and Configurations

We implemented BOCA in Python based on scikit-learn [51] and NumPy [52]. We adopted the implementation of RF provided in scikit-learn and used its default parameter settings. By conducting a preliminary study based on a small dataset, we set K, *offset*, *decay*, *scale* in BOCA to be 8, 20, 0.5, and 10, respectively. We set the initial set size to be 2 following the existing work [30] and the total number of iterations to be 60. We also investigated the influence of main parameters on the experimental results (in Section VI-A).

For the compared approaches, we adopted their implementations released by the corresponding work [10], [16], [29], [30], [53]. We also adopted the same parameter settings as the existing work [16], [29], [30], [53], and the same initial set size as BOCA for fair comparison (except GA). Due to crossover in GA, we cannot set the initial set size to 2, thus we set it to the closest value, i.e., 4. In the existing work [10], the initial set size of GA is set to 100, but it is very costly especially when the compilation and execution time of a program is large, thus we did not adopt this setting. To investigate the effectiveness of GA with a larger initial set size, we also tried another initial size of 10. We call the two GA implementations GA_4 and



(a) GCC (b) LLVM

Fig. 2: speedup of BOCA

GA_{10}, respectively. Following the existing work [17], we set n_b to be 10 and n_r to be 10,000 in BOCA$_s$.

To reduce the influence of machine environment, we ran a program under an optimization sequence 5 times and calculated the average time. To reduce the influence of randomness, we ran each approach 5 times and calculated the average results. Our study is conducted on a workstation with 16-core Intel(R) Xeon(R) CPU E5-2640 v3, 126G memory, and CentOS 6.10 operating system.

D. Measurement

Following the existing work [5], [25], we calculated the speedup of a compiler autotuning approach over `-O3` (i.e., the highest optimization level of GCC and LLVM with the performance goal of minimizing execution time of compiled programs [26]) to measure the effectiveness of an approach. The speedup is calculated by dividing the execution time of the program compiled under `-O3` by that of the program compiled under the optimization sequence produced by an approach. As our work aims to improve the efficiency of compiler autotuning, we compared these approaches in terms of the time spent on achieving (i.e., first time to reach or exceed) a certain speedup. Here, we set the total number of iterations for BOCA to be 60. We compared them in terms of the time they spent on achieving the speedup achieved by BOCA at its 30^{th}, 40^{th}, 50^{th}, 60^{th} iterations, respectively. Less is better. If a compared approach did not achieve a certain speedup within 120 iterations (2 times of the number of iterations of BOCA), we regarded it as *timeout*. In Section VI-B, we also discuss the effectiveness of BOCA with more iterations.

V. Results and Analysis

A. RQ1: BOCA v.s. Existing Compiler Autotuning Approaches

We first show the speedups of BOCA within 60 iterations over `-O3` on both GCC and LLVM in Figure 2. From this figure, BOCA indeed improves the runtime performance of compiled programs compared with the highest vendor-provided optimization level `-O3`. The average speedups of BOCA on GCC and LLVM are 1.25X and 1.13X respectively, demonstrating the effectiveness of BOCA in compiler autotuning. For example, for the program C5 on GCC, BOCA enables 44 optimization flags and disables the other 27 flags, which improves the runtime performance of C5 compiled under `-O3` by 1.63X. Also, compared with `-O3`, the average speedup of BOCA on GCC improves from 1.20X at the 20^{th} iteration to 1.25X at the 60^{th} iteration, and on LLVM it improves from 1.09X at the 20^{th} iteration to 1.13X at the 60^{th} iteration.

As BOCA is designed to improve the efficiency of compiler autotuning, we extensively compared BOCA with the existing compiler autotuning approaches (i.e., RIO, GA$_4$, GA$_{10}$, and Irace), to investigate how much time they spent in order to achieve the speedup achieved by BOCA. The results are shown in Table II, where the values represent the time spent on achieving the speedup achieved by BOCA at its 30^{th}, 40^{th}, 50^{th}, 60^{th} iterations, respectively. We also calculated the results at the 20^{th} iteration, which has the same conclusion as those at other iterations, and due to the space limit we put them at the project webpage. The cells with the shading ▨ refer to the best result in the corresponding case, and the cells marked as ✗ mean that the compared approach does not achieve the corresponding speedup within 120 iterations. Note that the time for completing 120 iterations for all the compared approaches is significantly longer than that for completing 60 iterations for BOCA. We calculated the average time spent on completing one iteration for each approach, which is 96, 102, 95, 100, and 136 seconds for BOCA, RIO, GA$_4$, GA$_{10}$, and Irace respectively, demonstrating the little cost of BOCA. Although running these compiler autotuning approaches takes some time, it is actually acceptable since the given program can be compiled once under the desired optimization sequence and then used all the time. From Table II, among 160 cases (20 programs * 2 compilers * 4 speedup settings), BOCA performs the best (i.e., requires the shortest time to achieve the corresponding speedup) among all the compared compiler autotuning approaches in 81% (129 out of 160) cases. Moreover, these compared approaches are even timeout in at least 45% cases. The results demonstrate that BOCA does largely improve the efficiency of compiler autotuning.

We further investigated whether BOCA significantly outperforms all the compared compiler autotuning approaches by conducting the Wilcoxon Signed-Rank Test [54] for their time spent on achieving the corresponding speedup at the significance level 0.05. For the timeout cases, we used the time spent on completing 120 iterations for statistical analysis and following calculation. We found that all the p values are smaller than 0.0004, indicating that BOCA significantly outperforms all the compared approaches for each speedup setting. We also calculated the average improvements of BOCA over the compared approaches in terms of the time they spent on achieving the speedup that BOCA achieved at its 30^{th}, 40^{th}, 50^{th}, 60^{th} iterations. The results are shown in Table III. The average improvements of BOCA over all the compared compiler autotuning approaches in terms of the time spent range from 42.30% to 78.04% across all speedup settings. That further confirms the effectiveness of BOCA.

B. RQ2: BOCA v.s. Existing Bayesian Optimization Methods

To answer RQ2, we compared BOCA with the direct application of existing Bayesian optimization methods (i.e., ε-PAL, FLASH, and TPE) in compiler autotuning. The comparison results between BOCA and the advanced method TPE are

TABLE II: Comparison among compared approaches in terms of time (seconds)

App.	ID	GCC				LLVM				ID	GCC				LLVM			
		S_{30}	S_{40}	S_{50}	S_{60}	S_{30}	S_{40}	S_{50}	S_{60}		S_{30}	S_{40}	S_{50}	S_{60}	S_{30}	S_{40}	S_{50}	S_{60}
BOCA	C1	3531	4839	6149	6589	3860	5430	7401	8781	P1	1785	2714	3780	4022	1933	2428	2428	3421
RIO		9095	13136	✗	✗	9783	11906	11906	15953		6053	✗	✗	✗	9173	✗	✗	✗
GA_4		17542	✗	✗	✗	16626	16837	16837	✗		3805	✗	✗	✗	✗	✗	✗	✗
GA_{10}		8497	12249	12746	12746	10573	12712	12712	13718		✗	✗	✗	✗	2437	✗	✗	✗
IRace		✗	✗	✗	✗	✗	✗	✗	✗		✗	✗	✗	✗	✗	✗	✗	✗
TPE		13601	✗	✗	✗	14035	15031	16138	18346		1117	4024	4651	7016	1165	4436	4436	4436
BOCA	C2	3363	4235	5753	7072	3279	4264	4912	6336	P2	3081	4278	4912	5987	2453	3506	3506	4939
RIO		5525	6737	8609	10693	9418	11977	11977	✗		✗	✗	✗	✗	920	7554	7554	13045
GA_4		10744	11142	✗	✗	12020	13232	14174	✗		4460	4460	✗	✗	4150	✗	✗	✗
GA_{10}		6254	6772	7748	8967	9959	11673	12077	✗		3025	3025	6661	✗	2301	5896	5896	8472
IRace		✗	✗	✗	✗	✗	✗	✗	✗		✗	✗	✗	✗	✗	✗	✗	✗
TPE		4603	4774	7066	9410	9488	9881	9931	10501		✗	✗	✗	✗	1521	7285	7285	9198
BOCA	C3	1942	2458	2757	3680	1931	2128	2864	4015	P3	4191	5894	7283	7283	5230	6910	7403	7403
RIO		✗	✗	✗	✗	✗	✗	✗	✗		9921	9921	9921	9921	15400	✗	✗	✗
GA_4		7175	✗	✗	✗	7977	✗	✗	✗		✗	✗	✗	✗	19891	✗	✗	✗
GA_{10}		4219	✗	✗	✗	6224	6224	6224	6224		✗	✗	✗	✗	5946	16894	16894	16894
IRace		✗	✗	✗	✗	✗	✗	✗	✗		✗	✗	✗	✗	✗	✗	✗	✗
TPE		6860	✗	✗	✗	✗	✗	✗	✗		✗	✗	✗	✗	✗	✗	✗	✗
BOCA	C4	2961	3986	4808	5853	2072	3504	3991	4955	P4	1231	1983	2476	3143	861	1782	2219	2219
RIO		✗	✗	✗	✗	7555	10161	10811	10811		✗	✗	✗	✗	758	1002	1002	1002
GA_4		9313	10012	11117	11117	6798	8008	8108	8108		✗	✗	✗	✗	1709	2298	✗	✗
GA_{10}		✗	✗	✗	✗	8181	9574	9574	9574		✗	✗	✗	✗	1409	2221	2636	2636
IRace		✗	✗	✗	✗	✗	✗	✗	✗		✗	✗	✗	✗	✗	✗	✗	✗
TPE		✗	✗	✗	✗	8074	✗	✗	✗		✗	✗	✗	✗	471	635	772	772
BOCA	C5	2464	3192	3715	4884	2369	3196	3920	4814	P5	5985	7469	8526	11439	8062	9059	12680	15258
RIO		7029	9355	9355	9693	7747	✗	✗	✗		11944	14293	23375	23375	56660	56660	56660	56660
GA_4		8603	9059	9059	9059	✗	✗	✗	✗		9740	13697	19294	20096	23278	✗	✗	✗
GA_{10}		5369	✗	✗	✗	6184	✗	✗	✗		16238	✗	✗	✗	40982	✗	✗	✗
IRace		1090	1090	1090	✗	✗	✗	✗	✗		✗	✗	✗	✗	✗	✗	✗	✗
TPE		5532	7303	8108	8809	7694	✗	✗	✗		✗	✗	✗	✗	20048	✗	✗	✗
BOCA	C6	1994	2585	3286	3816	1924	2554	3291	4009	P6	745	985	1463	1463	963	963	1637	1637
RIO		7070	✗	✗	✗	8404	8404	9420	9420		428	428	428	428	✗	✗	✗	✗
GA_4		5363	7207	7634	8059	4272	4272	6209	6209		144	144	144	144	✗	✗	✗	✗
GA_{10}		4576	7207	7207	7207	5359	5765	6585	6585		336	336	336	336	✗	✗	✗	✗
IRace		2903	✗	✗	✗	✗	✗	✗	✗		✗	✗	✗	✗	672	✗	✗	✗
TPE		5947	✗	✗	✗	9521	9521	✗	✗		150	150	265	265	✗	✗	✗	✗
BOCA	C7	2101	2851	3197	4615	2286	3064	3804	4466	P7	2823	3797	4617	5692	1730	2510	3157	3431
RIO		8939	✗	✗	✗	7024	7579	8114	10128		✗	✗	✗	✗	✗	✗	✗	✗
GA_4		6787	7797	8177	8465	2984	5888	6010	7388		6866	7367	7615	7615	✗	✗	✗	✗
GA_{10}		✗	✗	✗	✗	3096	6470	7488	8195		6129	✗	✗	✗	✗	✗	✗	✗
IRace		✗	✗	✗	✗	✗	✗	✗	✗		✗	✗	✗	✗	✗	✗	✗	✗
TPE		✗	✗	✗	✗	8375	✗	✗	✗		✗	✗	✗	✗	✗	✗	✗	✗
BOCA	C8	3557	4484	5882	6977	5017	6425	8189	8824	P8	1550	1831	2163	2163	1064	1064	1064	1064
RIO		12249	14329	✗	✗	17686	19047	✗	✗		✗	✗	✗	✗	949	949	949	949
GA_4		13395	17226	✗	✗	26282	✗	✗	✗		2246	✗	✗	✗	✗	✗	✗	✗
GA_{10}		11679	13164	13577	15017	22713	25011	✗	✗		✗	✗	✗	✗	1329	1329	1329	1329
IRace		✗	✗	✗	✗	✗	✗	✗	✗		✗	✗	✗	✗	✗	✗	✗	✗
TPE		9693	10885	12051	12142	17094	✗	✗	✗		4363	✗	✗	✗	659	659	659	659
BOCA	C9	1156	1637	2052	2453	2359	3264	3979	4524	P9	1824	2291	3007	3456	1754	1754	3071	3071
RIO		4712	✗	✗	✗	746	1897	2164	2866		4846	7267	✗	✗	✗	✗	✗	✗
GA_4		4047	4704	5385	5696	4728	5827	6364	6454		✗	✗	✗	✗	✗	✗	✗	✗
GA_{10}		1784	4430	6251	6709	1636	2092	2092	2642		2463	✗	✗	✗	✗	✗	✗	✗
IRace		1995	✗	✗	✗	✗	✗	✗	✗		✗	✗	✗	✗	✗	✗	✗	✗
TPE		✗	✗	✗	✗	6695	6853	6853	6853		✗	✗	✗	✗	✗	✗	✗	✗
BOCA	C10	1599	2200	2417	2916	1951	2565	2958	3836	P10	1450	1783	2701	3250	1008	1354	1998	2322
RIO		3312	4834	5559	✗	6149	6604	6735	6735		✗	✗	✗	✗	✗	✗	✗	✗
GA_4		4966	4966	4966	4966	✗	✗	✗	✗		970	970	✗	✗	✗	✗	✗	✗
GA_{10}		1607	1715	2137	2137	6535	6535	6950	6950		✗	✗	✗	✗	✗	✗	✗	✗
IRace		✗	✗	✗	✗	✗	✗	✗	✗		✗	✗	✗	✗	✗	✗	✗	✗
TPE		✗	✗	✗	✗	7424	7424	✗	✗		749	749	2506	3671	3749	3749	4926	4926

shown in Tables II and III (i.e., Row "TPE"). Table II shows that BOCA outperforms TPE in 88.75% (142 out of 160) cases. Table III shows that the average improvements of BOCA over TPE range from 43.01% to 71.06% across all speedup settings. We also conducted the Wilcoxon Signed-Rank Test between BOCA and TPE, and the results show that BOCA statistically significantly outperforms TPE.

Since ε-PAL and FLASH need to enumerate and predict all the unevaluated optimization sequences in each iteration and the number of optimization flags is large in our study, both of them cannot produce the results of compiler autotuning within acceptable time. As an example, considering the smallest program P4 with GCC, when tuning only 30 optimization flags, the time spent on completing the first iteration for both ε-PAL and FLASH (i.e., more than 13,000 seconds) has already been larger than the time spent on completing 60 iterations for BOCA (i.e., 3,143 seconds). Therefore, we cannot directly compare BOCA with ε-PAL and FLASH based on the whole

TABLE III: Average improvements of BOCA over compared approaches in terms of time spent on achieving speedups (%)

BOCA v.s.	GCC				LLVM			
	S_{30}	S_{40}	S_{50}	S_{60}	S_{30}	S_{40}	S_{50}	S_{60}
RIO	64.17	57.99	53.27	45.22	73.75	70.87	64.42	60.64
GA_4	66.37	60.33	56.37	48.36	71.41	69.12	62.48	56.54
GA_{10}	60.88	57.68	50.17	42.30	66.93	69.02	62.32	56.71
Irace	71.06	63.67	55.10	48.65	78.04	71.99	65.06	58.92
TPE	66.60	58.11	50.38	43.01	71.06	68.70	61.75	55.96

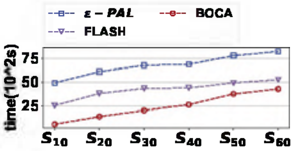


Fig. 3: BOCA vs ε-PAL vs FLASH

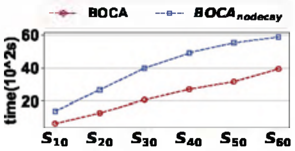


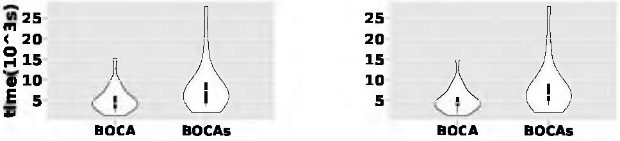
Fig. 4: BOCA vs $\text{BOCA}_{nodecay}$

set of flags used in our study. We further tried our best to enable comparisons by using a small set of optimization flags. More specifically, we conducted a study by randomly selecting 20 flags for GCC and LLVM respectively and randomly selecting four programs (i.e., C4, C7, P7, and P8) as the representatives. The results are shown in Figure 3, where the x-axis represents the speedups and the y-axis represents the time spent on achieving the speedups on average across programs and compilers. We found that even though using a small set of flags, BOCA still spends less time to achieve each specified speedup than both ε-PAL and FLASH.

In summary, our results confirm that BOCA indeed performs better than the existing Bayesian optimization methods, demonstrating the necessity of designing a novel Bayesian optimization method specific to compiler autotuning. Indeed, through RF, BOCA can use a relatively small training set to build an effective model.

C. RQ3: Selection Strategy Effectiveness

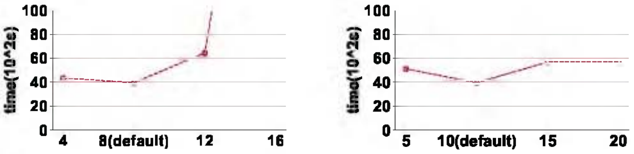
We further investigated whether or not our proposed selection strategy is more effective than the existing advanced selection strategy used in Bayesian optimization by comparing BOCA and BOCA_s. Figure 5 shows the comparison results in terms of the time spent on achieving the speedup achieved by BOCA in the 60^{th} iteration. We found that BOCA spends less time than BOCA_s on achieving the speedup on average for both GCC and LLVM. The average time spent by BOCA and BOCA_s is 4,838 and 8,412 seconds on GCC and 4,966 and 7,864 seconds on LLVM respectively. The average improvements of BOCA over BOCA_s across all programs are 42.49% and 36.45% on GCC and LLVM respectively. We also conducted the Wilcoxon Signed-Rank Test and found BOCA indeed significantly outperforms BOCA_s in statistics. The results demonstrate the effectiveness of our proposed selection strategy in BOCA. We further analyzed the reason why BOCA outperforms BOCA_s. The latter conducts local search by flipping only one flag and uses the acquisition function to select the final candidates while the former carefully measures the impact of each optimization flag and then exploits the identified impactful optimizations sufficiently. The results show that different optimization flags contribute differently to the overall effectiveness of compiler autotuning for a program.



(a) GCC

(b) LLVM

Fig. 5: Comparison between BOCA and BOCA_s



(a) k-value comparison

(b) scale comparison

Fig. 6: Parameter evaluation

As presented in Section III-D, to further improve the efficiency of compiler autotuning, we designed an exploration process with *decay*. It is also interesting to investigate whether such a *decay* process is really helpful to improve the efficiency. Therefore, we conducted a study to compare BOCA with its variant without the decay process (denoted as $\text{BOCA}_{nodecay}$) on the same programs used in Figure 3. The results are shown in Figure 4. BOCA indeed takes less time to achieve the same speedup than $\text{BOCA}_{nodecay}$ in all the settings, demonstrating the contribution of the *decay* process.

VI. Discussion

A. Influence of Main Parameters in BOCA

We discuss the influence of main parameters in BOCA based on the same programs used in Figure 3. In particular, we investigated two main parameters in BOCA: K (the number of identified impactful optimizations) and *scale* (which controls the speed of decay) that are described in Section III-D. The results are shown in Figure 6, where the x-axis represents the parameter values and the y-axis represents the average time spent on achieving the speedup BOCA achieves in the 60^{th} iteration across programs and compilers. From Figure 6a, our default K value, i.e., 8, performs the best. When K is set to 16, BOCA cannot be completed within the given time period, since exploiting all the combinations of 16 impactful optimizations is very costly. The result also demonstrates the importance of setting a proper K value. From Figure 6b, the small values (i.e., 5 and 10) of *scale* perform better than the large values (i.e., 15 and 20), indicating that relatively quick decay is helpful to improve the efficiency of compiler autotuning. Our current value of *scale* (i.e., 10) performs slightly better.

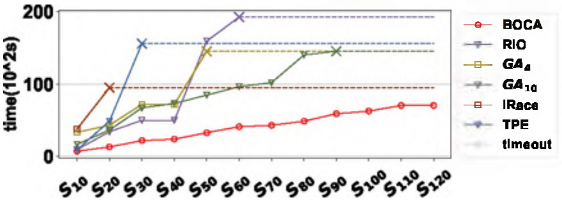


Fig. 7: Comparison with more iterations

B. Effectiveness of BOCA with More Iterations

We investigated the effectiveness of BOCA with more iterations by completing 120 iterations for BOCA and 240 iterations for the compared approaches. The results on program P7 on LLVM are shown in Figure 7, where the x-axis represents the speedups that BOCA achieves at the corresponding iterations, and the y-axis represents the time spent on achieving the speedups. From this figure, we can see that BOCA keeps outperforming all the compared approaches in terms of the time spent, with the number of iterations increasing. Moreover, the compared approaches are all timeout at a certain point (marked by × in Figure 7).

C. Threats to Validity

The threats to validity mainly lie in the compilers and programs used in our experiments. As presented in Section IV-A, to satisfy the prerequisites of the used benchmarks, we used relatively old versions of GCC and LLVM following existing work [5], [48]. To reduce the threats, we conducted an experiment to investigate the effectiveness of BOCA on a recent GCC version (i.e., GCC 8.3.1) on three different programs produced by Csmith [55]. Csmith is a state-of-the-art C program generator, and its produced programs tend to invoke a large amount of compiler-optimization code, which facilitates the evaluation of compiler autotuning. The results show that BOCA achieves 1.04X$\sim$1.06X speedups over `-O3` at the 60^{th} iteration on the three programs, while all the compared approaches achieve at most 1.01X speedup even at their 120^{th} iterations, demonstrating the effectiveness of BOCA on a recent compiler and on different programs to some degree. This speedup is relatively small since the performance of newer versions has been largely improved, which causes short execution time of the three small programs under `-O3`. Therefore, the improvement room for compiler autotuning becomes small on them. For larger programs, the improvement room could be larger. Due to space constraint, we only briefly describe this experiment here. More detailed results can be found in our project webpage. In the future, we will use more compilers and programs to evaluate the effectiveness of BOCA to further reduce the threats.

Furthermore, in our study, it is infeasible to know the ground-truth global optimum in an enormous search space. To further investigate whether BOCA is able to find the global optimum more efficiently, we found a small LLVM case which only has 11 flags and was completely searched [30]. We applied BOCA and the most effective compared approach TPE to it. BOCA found the global optimal sequence within 31 iterations while TPE cannot find it within 120 iterations, demonstrating the effectiveness of BOCA in finding the global optimal sequence to some degree.

VII. Conclusion

In this paper, we propose BOCA, the first Bayesian optimization based approach for efficient compiler optimization. BOCA includes a novel searching strategy for Bayesian optimization, which incorporates the impact of optimization flags measured by a tree-based model and a decay function. We perform extensive experiments on two widely-used C benchmarks using GCC and LLVM. The results demonstrate that BOCA significantly outperforms all the compared approaches in terms of the time spent on achieving specified speedups. Further, for compiler writers, our work will be useful for developing self-tuning compilers and improving compiler code associated with the flags that hamper performance.

Acknowledgment

This work has been supported by the National Natural Science Foundation of China 62002256 and 61828201.

References

[1] "GCC," Accessed: 2020, https://gcc.gnu.org.

[2] "LLVM," Accessed: 2020, https://llvm.org.

[3] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Computing Surveys*, vol. 53, no. 1, pp. 1–36, 2020.

[4] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, vol. 29, no. 12, pp. 1184–1201, 1986.

[5] Y. Chen, S. Fang, Y. Huang, L. Eeckhout, G. Fursin, O. Temam, and C. Wu, "Deconstructing iterative optimization," *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 3, pp. 1–30, 2012.

[6] 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. 96:1–96:42, 2019.

[7] A. H. Ashouri, G. Mariani, G. Palermo, and C. Silvano, "A bayesian network approach for compiler auto-tuning for embedded processors," in *12th IEEE Symposium on Embedded Systems for Real-time Multimedia*, 2014, pp. 90–97.

[8] F. V. Agakov, E. V. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams, "Using machine learning to focus iterative optimization," in *Fourth IEEE/ACM International Symposium on Code Generation and Optimization*, 2006, pp. 295–305.

[9] J. Cavazos, G. Fursin, F. V. Agakov, E. V. Bonilla, M. F. P. O'Boyle, and O. Temam, "Rapidly selecting good compiler optimizations using performance counters," in *Fifth International Symposium on Code Generation and Optimization*, 2007, pp. 185–197.

[10] U. Garciarena and R. Santana, "Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions," in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, 2016, pp. 1159–1166.

[11] L. P. Cáceres, F. Pagnozzi, A. Franzin, and T. Stützle, "Automatic configuration of gcc using irace," in *International Conference on Artificial Evolution (Evolution Artificielle)*. Springer, 2017, pp. 202–216.

[12] J. Mockus, *Bayesian approach to global optimization: theory and applications*. Springer Science & Business Media, 2012, vol. 37.

[13] M. P. Ranjit, G. Ganapathy, K. Sridhar, and V. Arumugham, "Efficient deep learning hyperparameter tuning using cloud infrastructure: Intelligent distributed hyperparameter tuning with bayesian optimization in the cloud," in *12th IEEE International Conference on Cloud Computing*, 2019, pp. 520–522.

[14] C. E. Rasmussen, "Gaussian processes in machine learning," in *Summer School on Machine Learning*. Springer, 2003, pp. 63–71.

[15] P. I. Frazier, "A tutorial on bayesian optimization," *CoRR*, vol. abs/1807.02811, 2018.

[16] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in neural information processing systems*, 2011, pp. 2546–2554.

[17] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration (extended version)," *Technical Report TR-2010–10, University of British Columbia, Computer Science, Tech. Rep.*, 2010.

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

[19] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 180–190.

[20] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering*, 2017, pp. 700–711.

[21] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and X. Bing, "Coverage prediction for accelerating compiler testing," *IEEE Transactions on Software Engineering*, 2018.

[22] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Test case prioritization for compilers: A text-vector based approach," in *2016 IEEE International Conference on Software Testing, Verification and Validation*, 2016, pp. 266–277.

[23] "cBench," Accessed: 2020, https://ctuning.org/wiki/index.php/CTools:CBench.

[24] "PolyBench," Accessed: 2020, https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[25] A. H. Ashouri, G. Mariani, G. Palermo, E. E. Park, J. Cavazos, and C. Silvano, "Cobayn: Compiler autotuning framework using bayesian networks," *ACM Transactions on Architecture and Code Optimization*, vol. 13, pp. 1–25, 06 2016.

[26] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. V. Bonilla, J. Thomson, C. K. I. Williams, and M. F. P. O'Boyle, "Milepost gcc: Machine learning enabled self-tuning compiler," *International Journal of Parallel Programming*, vol. 39, pp. 296–327, 2010.

[27] M. Kong, A. Pop, L.-N. Pouchet, R. Govindarajan, A. Cohen, and P. Sadayappan, "Compiler/runtime framework for dynamic dataflow parallelization of tiled programs," *ACM Transactions on Architecture and Code Optimization*, vol. 11, no. 4, pp. 1–30, 2015.

[28] E. Park, J. Cavazos, L.-N. Pouchet, C. Bastoul, A. Cohen, and P. Sadayappan, "Predictive modeling in a polyhedral optimization space," *International journal of parallel programming*, vol. 41, no. 5, pp. 704–750, 2013.

[29] M. Zuluaga, A. Krause, and M. Püschel, "ε-pal: an active learning approach to the multi-objective optimization problem," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 3619–3650, 2016.

[30] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel, "Finding faster configurations using flash," *IEEE Transactions on Software Engineering*, 2018.

[31] R. Benassi, J. Bect, and E. Vazquez, "Robust gaussian process-based global optimization using a fully bayesian expected improvement criterion," in *International Conference on Learning and Intelligent Optimization*. Springer, 2011, pp. 176–190.

[32] W.-Y. Loh, "Classification and regression trees," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 14–23, 2011.

[33] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 167–177.

[34] J. Guo, D. Yang, N. Siegmund, S. Apel, A. Sarkar, P. Valov, K. Czarnecki, A. Wasowski, and H. Yu, "Data-efficient performance learning for configurable systems," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1826–1867, 2018.

[35] H. Ha and H. Zhang, "Deepperf: performance prediction for configurable software with deep sparse neural network," in *2019 IEEE/ACM 41st International Conference on Software Engineering*. IEEE, 2019, pp. 1095–1106.

[36] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: static analyses and empirical results," in *36th International Conference on Software Engineering*, 2014, pp. 140–151.

[37] M. Lillack, C. Kästner, and E. Bodden, "Tracking load-time configuration options," *IEEE Trans. Software Eng.*, vol. 44, no. 12, pp. 1269–1291, 2018.

[38] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, E. D. Nitto, M. Harman, and P. Heymans, Eds. ACM, 2015, pp. 284–294.

[39] H. Ha and H. Zhang, "Performance-influence model for highly configurable software with fourier learning and lasso regression," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 470–480.

[40] Y. Shen, M. Seeger, and A. Y. Ng, "Fast gaussian process regression using kd-trees," in *Advances in neural information processing systems*, 2006, pp. 1225–1232.

[41] Z. Wang, F. Hutter, M. Zoghi, D. Matheson, and N. de Feitas, "Bayesian optimization in a billion dimensions via random embeddings," *Journal of Artificial Intelligence Research*, vol. 55, pp. 361–387, 2016.

[42] R. E. Banfield, L. O. Hall, K. W. Bowyer, and W. P. Kegelmeyer, "A comparison of decision tree ensemble creation techniques," *IEEE transactions on pattern analysis and machine intelligence*, vol. 29, no. 1, pp. 173–180, 2006.

[43] B. Xu, J. Z. Huang, G. Williams, Q. Wang, and Y. Ye, "Classifying very high-dimensional data with random forests built from small subspaces," *International Journal of Data Warehousing and Mining*, vol. 8, no. 2, pp. 44–63, 2012.

[44] B. H. Menze, B. M. Kelm, R. Masuch, U. Himmelreich, P. Bachert, W. Petrich, and F. A. Hamprecht, "A comparison of random forest and its gini importance with standard chemometric methods for the feature selection and classification of spectral data," *BMC bioinformatics*, vol. 10, no. 1, pp. 1–16, 2009.

[45] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, and L. Zhang, "History-guided configuration diversification for compiler test-program generation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering*, 2019, pp. 305–316.

[46] J. Chen, H. Ma, and L. Zhang, "Enhanced compiler bug isolation via memoized search," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2020, pp. 78–89.

[47] J. Chen, J. Han, P. Sun, L. Zhang, D. Hao, and L. Zhang, "Compiler bug isolation via effective witness test program generation," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 223–234.

[48] S. Purini and L. Jain, "Finding good optimization sequences covering program space," *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 4, pp. 1–23, 2013.

[49] J. T. Springenberg, A. Klein, S. Falkner, and F. Hutter, "Bayesian optimization with robust bayesian neural networks," in *Advances in neural information processing systems*, 2016, pp. 4134–4142.

[50] V. Dalibard, M. Schaarschmidt, and E. Yoneki, "Boat: Building auto-tuners with structured bayesian optimization," in *Proceedings of the 26th International Conference on World Wide Web*, 2017, pp. 479–488.

[51] "scikit-learn," Accessed: 2020, https://scikit-learn.org/stable/.

[52] "NumPy," Accessed: 2020, https://numpy.org.

[53] L. P. Cáceres, F. Pagnozzi, A. Franzin, and T. Stützle, "Automatic configuration of gcc using irace," in *International Conference on Artificial Evolution (Evolution Artificielle)*. Springer, 2017, pp. 202–216.

[54] F. Wilcoxon, S. Katti, and R. A. Wilcox, "Critical values and probability levels for the wilcoxon rank sum test and the wilcoxon signed rank test," *Selected tables in mathematical statistics*, vol. 1, pp. 171–259, 1970.

[55] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011, pp. 283–294.