Coverage Prediction for Accelerating Compiler Testing

Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, Bing Xie

Abstract—Compilers are one of the most fundamental software systems. Compiler testing is important for assuring the quality of compilers. Due to the crucial role of compilers, they have to be well tested. Therefore, automated compiler testing techniques (those based on randomly generated programs) tend to run a large number of test programs (which are test inputs of compilers). The cost for compilation and execution for these test programs is significant. These techniques can take a long period of testing time to detect a relatively small number of compiler bugs. That may cause many practical problems, e.g., bringing a lot of costs including time costs and financial costs, and delaying the development/release cycle. Recently, some approaches have been proposed to accelerate compiler testing by executing test programs that are more likely to trigger compiler bugs earlier according to some criteria. However, these approaches ignore an important aspect in compiler testing: different test programs may have similar test capabilities (i.e., testing similar functionalities of a compiler, even detecting the same compiler bug), which may largely discount their acceleration effectiveness if the test programs with similar test capabilities are executed all the time. Test coverage is a proper approximation to help distinguish them, but collecting coverage dynamically is infeasible in compiler testing since most test programs are generated on the fly by automatic test-generation tools like Csmith. In this paper, we propose the first method to predict test coverage statically for compilers, and then propose to prioritize test programs by clustering them according to the predicted coverage information. The novel approach to accelerating compiler testing through coverage prediction is called COP (short for COverage Prediction). Our evaluation on GCC and LLVM demonstrates that COP significantly accelerates compiler testing, achieving an average of 51.01% speedup in test execution time on the existing constructed dataset and achieving an average of 68.74% speedup on the new dataset including 12 latest release versions of GCC and LLVM. Moreover, COP outperforms the state-of-the-art acceleration approach significantly by improving 17.16%∼82.51% speedups in different settings on average.

Index Terms—Compiler Testing, Test Prioritization, Machine Learning

1 Introduction

It is well known that compilers are one of the most fundamental software systems and are crucial to the success of software-intensive projects. Compiler testing is an effective way to assure the quality of compilers [1]. Many compiler testing techniques have been proposed to automate compiler testing [1], [2], [3]. Due to the crucial role of compilers, they have to be well tested. Therefore, automated compiler testing techniques (those based on randomly generated programs by automatic test-generation tools like Csmith [2]) tend to run a large number of test programs. The cost for compilation and execution for these test programs is significant. These techniques can take a long period of testing time to detect a relatively small number of compiler bugs. For example, in the existing work [2], testing 1,000,000 random generated C programs by Csmith with 5 compiler configurations took about 1.5 weeks on 20 machines in the Utah Emulab testbed. Each machine had one quad-core Intel Xeon E5530 processor running at 2.4 GHz. That is, roughly 302 minutes are required to test 1000 programs generated by Csmith on one computer. Therefore, compiler testing suffers from the efficiency problem. Also, our industry partners point out emphatically the efficiency problem of compiler testing, since it causes many practical problems, e.g., bringing a lot of costs including time costs and financial costs, which are often unaffordable to companies, especially small ones, and delaying the development/release cycle. Therefore, accelerating compiler testing is necessary.

To accelerate compiler testing, some approaches have been proposed in recent years [4], [5]. These approaches utilize some criteria to prioritize test programs, so that test programs that are more likely to trigger compiler bugs are executed earlier. For example, the state-of-the-art approach, named LET [5], prioritizes test programs based on their bug-revealing probabilities per unit time predicted by historical bug information. However, all these acceleration approaches do not consider an important aspect in compiler testing, which is that different test programs may have similar test capabilities, i.e., testing similar functionalities of a compiler, even detecting the same compiler bug. For example, as reported in the existing work [6], during the testing for a GCC compiler, there are nearly 4,000 failing test programs but they just trigger 46 unique bugs. If the test programs with similar test capabilities are executed all the time, it may largely discount the acceleration effectiveness of these approaches.

It is scarcely possible to know which test programs have similar test capabilities before testing. The only possible

- *Junjie Chen, Dan Hao, Yingfei Xiong, Lu Zhang, and Bing Xie are with Institute of Software, EECS, Peking University, Beijing, 100871, China and Key Laboratory of High Confidence Software Technologies (Peking University), MoE.*
  E-mail: {chenjunjie,haodan,xiongyf,zhanglucs,xiebing}@pku.edu.cn.
- *Guancheng Wang is with Jilin University, Changchun, 130012, China.*
  E-main:amocywang@gmail.com.
- *Hongyu Zhang is with The University of Newcastle, NSW 2308, Australia.*
  E-main:hongyu.zhang@newcastle.edu.au.

direction is to find a proper approximation to distinguish them. Test coverage (e.g., statement, method, and file coverage) is one of the most widely-used methods to measure test effectiveness, and it can be acquired in advance at the regression scenario. Also, it has been widely adopted by researchers and practitioners to improve software testing process [7], [8], [9], [10]. For example, during the process of regression test-suite reduction [11], [12], test coverage is widely used to measure the redundancy of tests. Therefore, test coverage may be used as a proper approximation to help identify which test programs have similar test capabilities. Intuitively, if the coverage information of test programs is very different, they tend to have different test capabilities.

Traditionally, coverage information is obtained by instrumenting programs at a series of locations (e.g., at the entry and exit points of methods) and executing the instrumented programs. The traditional method tends to be infeasible for compilers. This is because most test programs used in compiler testing are generated on the fly by automatic test-generation tools like Csmith, and thus the coverage information of these test programs is not available in advance. Also, due to their size and complexity, the process of collecting coverage is time-consuming, and the volume of the coverage information is also very large, which incurs overhead in storage and maintenance.

To acquire test coverage in advance, we propose the first method to predict test coverage statically for compilers (without executing the test programs). Based on the predicted coverage information, we propose to utilize clustering to distinguish test programs with different test capabilities, to accelerate compiler testing. That is, in this paper we propose a novel approach to accelerating compiler testing through coverage prediction. We call this approach COP[1] (short for **CO**verage **P**rediction). More specifically, COP first learns a coverage prediction model based on the historical coverage information and test program data. The key insight is that test programs with some specific language features, operation features, and structure features are more likely to cover some specific code regions of a compiler. Then, COP utilizes the learnt model to predict the coverage of each compiler module (e.g., source file or method) for each new test program. Next, based on the predicted coverage information, COP clusters test programs into different groups, and different groups are more likely to have different test capabilities. Finally, COP prioritizes test programs by enumerating each group to select test programs based on the bug-revealing probability per unit time of each test program, which is predicted by LET.

To evaluate the acceleration effectiveness of COP, we use two widely-used C compilers, i.e., GCC and LLVM, as subjects following the existing work on compiler testing [1], [2], [3], [4], [5]. In particular, We use the constructed dataset in the existing work [5], which is convenient to compare with the state-of-the-art approach LET. Since this dataset contains only old release versions of GCC and LLVM, we further construct a new dataset including 12 latest release versions of GCC and LLVM to sufficiently evaluate the acceleration effectiveness of COP. Our experimental results demonstrate that COP accelerates compiler testing in 93.97% cases and achieves an average of 51.01% speedup in test execution time on the existing constructed dataset, and achieves an average of 68.74% speedup on the dataset of latest release versions. Compared with LET, the state-of-the-art approach to accelerating compiler testing [5], COP outperforms it significantly by improving 17.16% ∼82.51% speedups in different settings on average. Furthermore, we also investigate the accuracy of COP on predicting coverage using GCC and LLVM. Our experimental results confirm the effectiveness of COP on coverage prediction: it achieves extremely small mean absolute errors, i.e., only 0.034∼0.051 when predicting coverage within the same version and 0.052∼0.116 when predicting coverage across the versions[2].

Our novelty can be summarized into three parts: identifying features, predicting coverage, applying predicted coverage to accelerate compiler testing. First, we define three categories of features that can characterize the coverage information through the understanding of existing compiler bugs and the characteristics of compiler inputs. Second, we propose the first method to predict coverage statically for compilers based on the historical coverage information and test-program features. Third, we propose a novel approach, COP, to accelerating compiler testing based on the predicted coverage information. Furthermore, we conduct an extensive experimental study confirming the effectiveness of COP, which significantly outperforms the state-of-the-art approach (LET) for accelerating compiler testing. It is also a contribution of this paper.

The structure of the paper is organized as follows. Section 2 introduces background about compiler testing techniques, compiler testing acceleration, and clustering; Section 3 presents our approach, COP; Section 4 presents the experimental study design; Section 5 presents the results and analysis of our experimental study; Section 6 presents the threats in our experimental study; Section 7 presents some discussions about this paper; Section 8 introduces some related work to this paper; Section 9 concludes this work.

2 Background

2.1 Compiler Testing Techniques

To guarantee the quality of compilers, many compiler testing techniques have been proposed in the literature. Chen et al. [1] conducted an empirical study to compare three mainstream compiler testing techniques. Here we introduce these techniques as follows. More related work about compiler testing techniques can be found in Section 8.1.

Randomized differential testing (RDT) [13] is a widely-used compiler testing technique in practice [2], [14]. RDT detects compiler bugs by using two or more comparable compilers that implement the same specification. More specifically, if using the same test programs as the inputs of these comparable compilers, they should produce the same results. Otherwise, compiler bugs are detected by

1. An original meaning of COP is a police officer who finds criminals by utilizing some trace information. Similarly, our approach, COP, also utilizes trace information (i.e., coverage information) to accelerate compiler bug detection.

2. The mean absolute error is the mean value of absolute differences between predicted values and true values. The range of mean absolute error is between zero and one, since each value is the covered fraction of a compiler module as presented in Section 3.

RDT. When the number of comparable compilers is larger than 2, RDT can determine which compiler contains the bugs through voting. The overview of RDT is shown in Figure 1(a), where $C_1, C_2, \ldots, C_n$ represents n comparable compilers, $O_1, O_2, \ldots, O_n$ represents the corresponding outputs produced by these compilers given a test program P and P's input I.

Different optimization levels (DOL) [1] is a variant of RDT. Different with RDT, DOL detects compiler bugs by using different optimization levels of a compiler. That is, when using different optimization levels of a compiler to compile and execute the same test programs, they should produce the same results. Otherwise, compiler bugs are detected by DOL. The overview of DOL is shown in Figure 1(b), where $L_1, L_2, \ldots, L_n$ represents n optimization levels of the compiler C, $O_1, O_2, \ldots, O_n$ represents the corresponding outputs produced by these optimization levels given a test program P and P's input I.

Equivalence modulo inputs (EMI) [3], [15], [16] is also an effective compiler testing technique. EMI first generates some equivalent variants under a set of test inputs for any given test program, and then detects compiler bugs by comparing the results produced by the pair of the original test program and its variants. If they produce different results, compiler bugs are detected by EMI. The overview of EMI is shown in Figure 1(c), where $V_1, V_2, \ldots, V_n$ represents n equivalent variants with the original test program P under the set of P's input I, $O_1, O_2, \ldots, O_n$ represents the corresponding outputs of variants produced by the compiler C, and O_p represents the output of P produced by C.

2.2 Compiler Testing Acceleration

To accelerate compiler testing, some approaches have been proposed recently [4], [5], including TB−G and LET.

TB−G is a text-vector based test-program prioritization approach [4]. It regards each test program as text, and extracts the tokens reflecting bug-relevant characteristics in order to transform each test program into a text-vector. In particular, the bug-relevant characteristics contain statement characteristics, type and modifier characteristics, and operator characteristics. After getting a set of text-vectors, TB−G normalizes the values of elements in vectors into an interval between zero and one. Finally, TB−G calculates the distance between normalized text-vectors and the origin vector $(0, 0, \ldots, 0)$ to prioritize test programs. More specifically, TB−G prioritizes test programs based on the descending order of distances.

LET is the state-of-the-art compiler testing acceleration approach [5]. It prioritizes test programs based on the historical bug information. More specifically, LET consists of a learning process and a scheduling process. In the learning process, LET first identifies bug-revealing features from test programs, and then preprocesses features of the training set, including feature selection and normalization. Next, LET builds a capability model, which is used to predict the bug-revealing probability for each new test program, and also builds a time model, which is used to predict the execution time of each new test program. In the scheduling process, LET uses the two models to predict the bug-revealing probability and execution time for each new test program, and then prioritizes test programs based on the descending order of their bug-revealing probabilities per unit time. Since the existing study [5] has demonstrated that LET performs significantly better than TB−G, in this paper we regard LET as the comparative acceleration approach.

Traditional test prioritization approaches are also discussed in the existing work [5]. However, these traditional test prioritization approaches either have bad effectiveness or cannot be applied to accelerate compiler testing. For example, the existing work [5] evaluated the adaptive random prioritization. It selects the test input that has the maximal distance to the set of already selected test inputs [17]. When adopting it to accelerate compiler testing, we treat test programs as test inputs, and calculate the distance between test programs using their edit distance. The evaluation shows that adaptive random prioritization has bad effectiveness in accelerating compiler testing due to extremely long time spent on prioritization.

2.3 Clustering

Clustering is a task to divide the data points into a number of groups such that data points in the same groups are more similar to those in other groups [18]. In other words, its aim is to segregate groups with similar properties and assign them into clusters. Over decades' developments, there are a lot of clustering algorithms, which follow various rules to define the similarity among data points, including connectivity models, centroid models, distribution models, and density Models [18]. In our approach, we use clustering to divide test programs into many groups, and different groups are more likely to have different test capabilities. Since we cannot know the number of groups before testing, the clustering algorithms that need to predefine the number of clusters cannot be applied to our problem. In particular, we use the X-means algorithm [19] in our approach. The X-means algorithm is a fast algorithm that estimates the number of clusters in K-means [20]. More specifically, it starts by setting the number of clusters to be equal to the lower bound of the given range, and continues to add centroids where they are needed until the upper bound is reached. During the process, the centroid sets achieving the best results are the final outputs.

3 Approach

COP consists of three steps: predicting coverage for test programs (see Section 3.1), clustering test programs (see Section 3.2), and prioritizing test programs (see Section 3.3).

3.1 Predicting Coverage

Predicting coverage is to predict the covered fraction of each compiler module (e.g., source file or method) by a test program. This can be regarded as a multi-output regression problem, where each label refers to the covered fraction of each compiler module. Similar to the typical process of machine learning, the step of predicting coverage contains identifying features, labeling, building a prediction model, and predicting and aligning coverage.

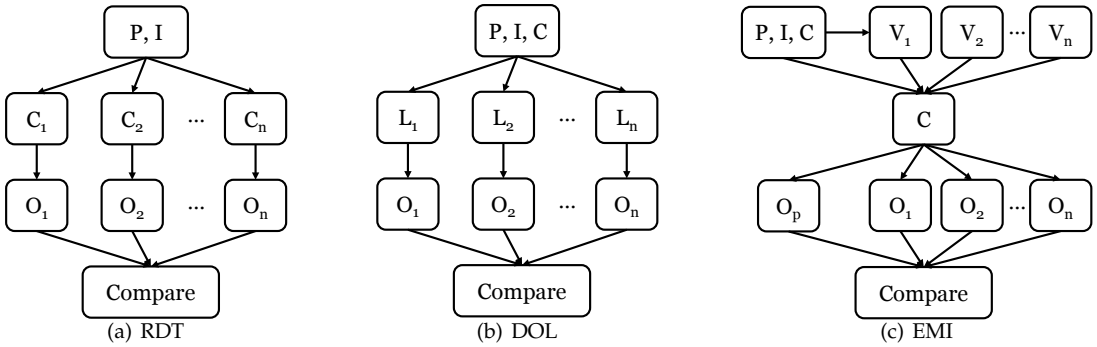


Fig. 1. Overview of three mainstream compiler testing techniques

3.1.1 Identifying Features

The key insight of predicting coverage is that test programs with some specific language features, operation features, and structure features are more likely to cover some specific code regions of compiler modules. If we can correctly identify these features from test programs, we should be able to predict the covered fraction of each compiler module by a test program before running it.

It is possible for us to identify features from test programs to predict coverage for compilers. For example, for the compiler modules in the front end, which is responsible to verify syntax and semantics, it is easy to predict their covered fractions. This is because some specific basic language features tend to directly reflect the covered fractions of these compiler modules. Similarly, for the compiler modules in the middle end and back end, which are responsible to conduct analysis, optimizations, and transformations on the different levels of code respectively, the existence of some language elements, and some specific operations and structures in a test program reflect whether some code regions of compiler modules in the two ends may be covered to some degree. For example, if multi-nested "for" loops exist in a test program, the code regions of loop optimizations are more likely to be covered by the test program.

More formally, the features used in COP are divided into three categories. The first category of features, language features, are concerned with whether the basic language elements exist in a test program. Language features are defined as three sets: $\mathcal{LANGUAGE} = \mathcal{STMT} \cup \mathcal{EXPR} \cup \mathcal{VAR}$, where $\mathcal{STMT}$ is the set of all statement types, $\mathcal{EXPR}$ is the set of all expression types, $\mathcal{VAR}$ is the set of all variable types in C language. Note that if a basic language element occurs at least one time, the value of the feature is set to one, otherwise it is set to zero.

The second category of features, operation features, are concerned with whether the basic operations exist in a test program and how complex operations are used. The basic operation features are the set of all operation types in C languages. The complex operation features include that:

- Taking address features, e.g., the number of times the address of a variable is taken.
- Comparing pointer features, e.g., the number of times a pointer is compared with the address of another variable or another pointer.
- Reading/writing features, e.g., the number of times a volatile/non-volatile variable is read/written.
- Pointing features, e.g., the number of pointers pointing to pointers/scalars/structs.
- Dereferencing pointer features, i.e., the times of a pointer is dereferenced on LHS/RHS.
- Jumping features, i.e., the times of forward jumps and backward jumps.

Note that, for basic operation features, if a basic operation occurs at least one time, the value of the feature is set to one, otherwise it is set to zero.

The third category of features, structure features, are concerned with the time of the occurrence of certain structures, and the depth of certain structures. More specifically, structure features include that:

- Block depth features, i.e., the max depth of blocks and the number of blocks with some specific depth.
- Struct depth features, i.e., the max depth of structs and the number of structs with some specific depth.
- Expression depth features, i.e., the max depth of expressions and the number of expressions with some specific depth.
- Bitfield of structs features, i.e., the number of structs with non-zero, zero, const, violate, full bitfields.

Our feature set is designed through a systematic exploration of the C syntax and the Csmith design. The language features and basic operation features cover all syntactic types in C. The complex operation features and structure features are reused from Csmith. These features are considered bug-relevant and are collected during the generation process of Csmith. Reusing these features also shortens the process, i.e., we do not have to collect these features again on the input programs.

3.1.2 Labeling

In our multi-output regression problem, we collect the covered fraction of each compiler module as each label. That is, the value of each label is a real number between zero and one. Since some compiler modules have quite similar covered code regions for almost all test programs (e.g., some compiler modules in the parser), the predicted results for these labels are meaningless, and thus we remove them from the label set. That is, we do not need to predict coverage for these compiler modules in our approach, since they cannot help distinguish test programs with different test

capabilities. Here we set that if there are more than 99% test programs whose absolute differences of coverage on a compiler module with the medium coverage of the set of test programs are less than 5%, we remove this compiler module from the label set.

3.1.3 Building a Prediction Model

We collect a set of training test programs generated by Csmith, and features and labels of each test program[3]. Based on the set of training instances (including features and labels), we first normalize features so as to adjust values measured on different scales to a common scale. Since the features are either numeric type or boolean type (i.e., 0 or 1), COP normalizes each value of these features into the interval [0, 1] using min-max normalization [21]. Suppose the set of training instances is denoted as $T = \{t_1, t_2, \ldots, t_m\}$ and the set of features is denoted as $F = \{f_1, f_2, \ldots, f_s\}$, we use a variable x_{ij} to represent the value of the feature f_j for the test program t_i before normalization and use a variable x^*_{ij} to represent the value of the feature f_j for the test program t_i after normalization ($1 \leq i \leq m$ and $1 \leq j \leq s$). The normalization formula is as follows:

$$x^*_{ij} = \frac{x_{ij} - min(\{x_{kj} | 1 \leq k \leq m\})}{max(\{x_{kj} | 1 \leq k \leq m\}) - min(\{x_{kj} | 1 \leq k \leq m\})}$$

After normalization, we then adopt the gradient boosting for regression [22] to build a prediction model for our multi-output regression problem. It builds an additive model in a forward stage-wise fashion, and it allows for the optimization of arbitrary differentiable loss functions. In each stage a regression tree is fit on the negative gradient of the given loss function. Its effectiveness to solve regression problems has been demonstrated by the existing work [22].

3.1.4 Predicting and Aligning Coverage

To predict coverage for each new test program, we also extract features for each of them. Then, based on the learnt prediction model, COP predicts the covered fraction of each remaining compiler module for each new test program.

Since COP is not limited to predict coverage for the same version of a compiler as the version used to collect labels, there may exist the label alignment problem when the version used to train a prediction model is different from the version to be predicted. The label alignment problem is that the labels collected on the two versions are not totally the same, due to the existence of newly added compiler modules and out-of-date compiler modules between versions. To solve the label alignment problem after prediction, we match compiler modules between versions based on the names of compiler modules, e.g., file names or method names. If the name of a compiler module stays the same between the two versions, the compiler module is regarded as a "matched" module. Otherwise, it is regarded as a "non-matched" module. In particular, we remove all "non-matched" modules to align coverage.

3. As the training test programs are generated by Csmith, we directly extract the values of features (including the three types of features) from each test program during the test-generation process of Csmith.

3.2 Clustering Test Programs

After acquiring the coverage information of new test programs, COP clusters them into different groups based on their predicted coverage information, and different groups are more likely to have different test capabilities. More specifically, COP uses the X-means clustering algorithm with Manhattan distance [19] for clustering, which does not require to predefine the number of clusters and estimates the number of clusters quickly by making a local decision after each run of K-means.

3.3 Prioritizing Test Programs

Since the test programs in different groups tend to have different test capabilities, COP prioritizes new test programs considering the clustering results. Before prioritization, COP first utilizes LET to acquire the bug-revealing probability per unit time for each new test program. Then, during prioritization, COP enumerates each group to select the test program with the largest bug-revealing probability per unit time in each group and ranks them, until all the test programs are selected. In particular, if bug-revealing probabilities per unit time of test programs are smaller than a threshold, denoted as ψ, COP does not select them until all test programs whose bug-revealing probabilities per unit time are larger than ψ are selected. The reason is that the test program with extremely small bug-revealing probabilities per unit time is unlikely to trigger compiler bugs.

Algorithm 1 formally depicts our prioritization strategy. Given a set of new test programs $T = \{t_1, t_2, \ldots, t_M\}$, which is clustered into the following groups $\{g_1, g_2, \ldots, g_N\}$, Algorithm 1 outputs an execution order of these test programs, which is recorded by a list $Rank$. In this algorithm, Lines 13 to 20 present the main procedure of our prioritization strategy. Lines 15 to 18 separate each group into two groups: $high$ and low. The former contains the test programs whose bug-revealing probability per unit time are larger than ψ, while the latter contains the remaining test programs. Lines 19 to 20 prioritize the test programs in $high$ and low by calling function $Prioritize$, respectively. In the $Prioritize$ function, Line 2 judges whether all test programs in $group$ have been prioritized. Lines 3 to 10 selects the test program with the largest bug-revealing probability per unit time from each non-empty group, and then removes it from the group. Line 11 sorts these selected test programs and records them into $Rank$.

4 Experimental Study Design

In this study, we address the following two research questions.

- **RQ1**: Does COP accelerate C compiler testing?
- **RQ2**: How does COP perform in terms of predicting coverage for C compilers?

We first evaluate whether COP can accelerate compiler testing, which is the ultimate goal of COP. Afterwards, we then investigate the accuracy of COP on predicting coverage.

Algorithm 1: Prioritizing Test Programs

```
Function Prioritize (group)
    while ∃k∈1...N group[k] is not empty do
        Selected← [ ]
        for j ← 1 to N by 1 do
            if group[j] is not empty then
                Max[j]←max(group[j])
                Selected←Selected∪{Max[j]}
                group[j]←group[j]-{Max[j]}
            end
        end
        Rank.add(Sort(Selected))
    end
Procedure Main()
    Rank← [ ]
    for j ← 1 to N by 1 do
        high[j]←{x∈ g[j] | x> ψ}
        low[j]←g[j]-high[j]
    end
    Prioritize(high)
    Prioritize(low)
```

TABLE 1
Subject statistics

Subject	LOC	#File	#Method	Usage
GCC-4.3.0	405,415	333	47,354	RQ1&RQ2
GCC-4.4.0	460,989	352	21,833	RQ2
GCC-4.4.3	461,688	352	21,894	RQ1
GCC-4.5.0	492,561	372	23,767	RQ2
GCC-4.6.0	525,010	390	25,168	RQ2
GCC-6.1.0	538,413	516	33,247	RQ1
GCC-6.2.0	538,368	516	33,250	RQ1
GCC-6.3.0	538,933	516	33,308	RQ1
GCC-6.4.0	539,279	516	33,318	RQ1
GCC-7.1.0	595,610	541	35,062	RQ1
GCC-7.2.0	595,824	541	35,073	RQ1
GCC-7.3.0	596,022	541	35,109	RQ1
LLVM-2.6	387,493	1,158	33,356	RQ1&RQ2
LLVM-2.7	462,188	1,350	37,682	RQ1&RQ2
LLVM-2.8	528,727	1,499	42,872	RQ2
LLVM-4.0.0	366,271	1,272	271,805	RQ1
LLVM-4.0.1	364,126	1,272	271,865	RQ1
LLVM-5.0.0	387,721	1,365	287,654	RQ1
LLVM-5.0.1	387,787	1,365	287,664	RQ1
LLVM-6.0.0	415,732	1,422	301,692	RQ1

4.1 Subjects and Test Programs

In this study, we use the two most widely-used C compilers, i.e., GCC and LLVM for the x86 64-Linux platform, which cover almost all C compilers used in the study of C compiler testing [1], [2], [3], [4], [14], [23]. More specifically, we use the constructed dataset (i.e., GCC-4.4.3, LLVM-2.6, and LLVM-2.7) in the existing work [5] to evaluate the effectiveness of COP on accelerating compiler testing, which is convenient to compare with the state-of-the-art acceleration approach LET. Since this dataset contains only old release versions of compilers, we further construct a new dataset that includes 12 latest release versions of GCC and LLVM (i.e., 7 GCC latest release versions, i.e., GCC-6.1.0, GCC-6.2.0, GCC-6.3.0, GCC-6.4.0, GCC-7.1.0, GCC-7.2.0, and GCC-7.3.0, and 5 LLVM latest release versions, i.e., LLVM-4.0.0, LLVM-4.0.1, LLVM-5.0.0, LLVM-5.0.1, and LLVM-6.0.0) to sufficiently investigate the effectiveness of COP. Furthermore, based on the dataset [5], we also additionally use more release versions of GCC and LLVM including GCC-4.3.0, GCC-4.4.0, GCC-4.5.0, GCC-4.6.0, and LLVM-2.8 to sufficiently evaluate the effectiveness of COP on predicting coverage.

Table 1 shows the statistical information of subjects. In this table, the first column presents the used compiler and each used specific release version; Columns 2-4 present the number of lines of code, the number of files, and the number of methods in each used release version of each compiler, respectively. In particular, these data refer to those used to be collected coverage in our study. For GCC, we collect coverage for all ".c" and ".h" files, while for LLVM, we collect coverage for all ".cpp" and ".h" since the part for compiling C programs of GCC is implemented using C while LLVM is implemented using C++. The last column presents the usage of each subject, i.e., used in RQ1 or RQ2.

The test programs used in our study are C programs randomly generated by Csmith [2], which is widely used in the literature of C compiler testing [1], [3], [4]. Each C program generated by Csmith is valid and does not require external inputs. The output of each generated program is the checksum of the non-pointer global variables of the program at the end of program execution. In particular, we generate 2,000 test programs using Csmith for GCC and LLVM as the training set for COP, respectively.

4.2 Implementations

In this study, we set each compiler module as each source file, considering the balance between efficiency and effectiveness. That is, COP accelerates compiler testing by predicting file coverage. In particular, we use Gcov[4] to collect file coverage in this study. Besides, the used gradient boosting for regression in COP is implemented in the scikit-learn[5], which is a simple and efficient tool for data mining and data analysis in python. The used X-means clustering algorithm is implemented in Weka 3.6.12 [24], which is a popular environment for data mining in Java. We use the gradient boosting for regression by setting $n\_estimators = 200$, $max\_depth = 5$, and $learning\_rate = 0.01$, and use the X-means clustering algorithm by setting the minimum number of clusters to 5, the maximum number of clusters to 50, and the maximum number of overall iterations to 10. The parameter values are decided by a preliminary study that we conduct on a small dataset. Other parameters in these two algorithms are set to the default values provided by scikit-learn and Weka, respectively. For the prioritization strategy presented in Section 3.3, we set ψ to the bug-revealing probability per unit time of the test program that are ranked at the 2/3 position in the ranking list of test programs.

4.3 Independent Variables

In this study, we consider the following three independent variables.

4. http://ltp.sourceforge.net/coverage/gcov.php.
5. http://scikit-learn.org/stable/

4.3.1 Application Scenarios

COP has two application scenarios, i.e., same-version scenario and cross-version scenario.

Same-version scenario means that COP trains a prediction model based on a version of a compiler, and then accelerates testing of the same version by using the prediction model to predict test coverage on this version. To evaluate COP in this application scenario, for the dataset [5] we use GCC-4.4.3, LLVM-2.6, and LLVM-2.7 to train the prediction model and use the model to accelerate testing of GCC-4.4.3, LLVM-2.6, and LLVM-2.7, respectively. We denote them as GCC-4.4.3→GCC4.4.3, LLVM-2.6→LLVM2.6, and LLVM-2.7→LLVM2.7. Besides, we also evaluate the effectiveness of COP on the new dataset including 12 latest release versions of GCC and LLVM in this application scenario.

Cross-version scenario means that COP trains a prediction model based on a version of a compiler, and accelerates testing of a later version of the compiler by using the prediction model to predict test coverage on the later version of the compiler. To evaluate the acceleration effectiveness of COP in this application scenario, for the dataset [5] we use GCC-4.3.0 and LLVM-2.6 to train a prediction model and use the model to accelerate testing of GCC-4.4.3 and LLVM-2.7, respectively. Since Clang is not integrated into LLVM before LLVM-2.6, we cannot use COP to accelerate testing of LLVM-2.6 in this application scenario. We denote them as GCC-4.3.0→GCC4.4.3 and LLVM-2.6→LLVM2.7.

Furthermore, we also additionally use more release versions of GCC and LLVM so as to sufficiently evaluate the accuracy of COP on predicting coverage. Therefore, in the same-version application scenario, we also use GCC-4.3.0, GCC-4.4.0, GCC-4.5.0, GCC-4.6.0, and LLVM-2.8 to train a prediction model, and use the prediction model to predict coverage on GCC-4.3.0, GCC-4.4.0, GCC-4.5.0, GCC-4.6.0, and LLVM-2.8, respectively. We denote them as GCC-4.3.0→GCC4.3.0, GCC-4.4.0→GCC4.4.0, GCC-4.5.0→GCC4.5.0, GCC-4.6.0→GCC4.6.0, and LLVM-2.8→LLVM-2.8. In the cross-version application scenario, we add GCC-4.3.0→GCC4.4.0, GCC-4.4.0→GCC4.5.0, GCC-4.5.0→GCC4.6.0, and LLVM-2.7→LLVM-2.8.

4.3.2 Compiler Testing Techniques

In this study, we consider two compiler testing techniques to accelerate following the existing work [5], i.e., Different Optimization Level (DOL) [1] and Equivalence Modulo Inputs (EMI) [3]. Since both RDT and DOL are differential-testing based compiler testing techniques as presented in Section 2.1, following the existing work [5] we use DOL as the representative in this paper.

DOL detects compiler bugs by comparing the results produced by the same test program with different optimization levels (i.e., -O0, -O1, -Os, -O2 and -O3) [1]. Given an execution order decided by an acceleration approach, we compile and execute test programs under different optimization levels, and determine whether the test program triggers a bug by comparing their results.

EMI detects compiler bugs by generating some equivalent variants for any given test program and comparing the results produced by the original test program and its variants[6] [3]. Given an execution order decided by an acceleration approach, we generate eight variants for each original test program by randomly deleting its unexecuted statements same as the EMI work [3], and then compile and execute each pair of a test program and its variants under the same optimization level (i.e., -O0, -O1, -Os, -O2 and -O3). Finally, we compare the corresponding results to determine whether a bug is detected by them.

4.3.3 Compared Approaches

In the study, to evaluate the acceleration effectiveness of COP, we have the following compared approaches.

Random order (RO), is taken as the baseline in this study. RO randomly determines an execution order of new test programs, demonstrating the effectiveness of compiler testing without any acceleration approaches.

LET [5], is the state-of-the-art acceleration approach. It predicts bug-revealing probability and execution time for each new test program based on historical bug information, and then prioritizes new test programs based on their bug-revealing probabilities per unit time.

Besides, to investigate the accuracy of COP on predicting coverage, we also have a baseline, i.e., random guess (RG), for coverage prediction. More specifically, RG randomly guesses the covered fraction of each predicted source file by a test program, i.e., a real number between zero and one.

4.4 Dependent Variables

To measure the effectiveness of acceleration approaches, following the existing work [5], we use the time spent on detecting r bugs, where $1 \leq r \leq u$ and u is the total number of bugs detected when executing all test programs in this study. Smaller is better. All these acceleration approaches, i.e., LET and COP, take extra time on scheduling test programs, and thus the time spent on detecting r bugs also includes the scheduling time. For example, the extra time of COP on scheduling test programs includes the time spent on predicting coverage, clustering and prioritizing test programs.

Based on it, we further use Formula 1 to calculate the corresponding speedup on detecting r bugs, where $TRO(r)$ refers to the time spent on detecting r bugs without any accelerating approaches and $TACC(r)$ refers to the time spent on detecting r bugs with one acceleration approach. Larger is better.

$$Speedup(r) = \frac{TRO(r) - TACC(r)}{TRO(r)} \quad (1)$$

Besides, we also investigate the accuracy of COP on predicting coverage. Here we adopt the commonly-used metrics in the multi-output regression problem, i.e., explained variance score [25], mean absolute error [26], and R^2 score [27].

The explained variance score (EV) measures the proportion to which a mathematical model accounts for the variation of a given data set. Supposed that $\hat{y}$ is the predicted target output, y is the corresponding true target output, and

6. In fact, EMI has three instantiations, namely Orion [3], Athena [15], and Hermes [16]. In our paper, EMI refers to Orion.

Var refers to the variance, the explained variance score is computed following Formula 2. Larger is better.

$$Expected\_variance(y, \hat{y}) = 1 - \frac{Var(y - \hat{y})}{Var(y)} \quad (2)$$

The mean absolute error (MAE) refers to the mean value of absolute differences between predicted values and true values. Supposed that $\hat{y}_i$ is the predicted value of the ith output, y_i is the corresponding true value, and n refers to the number of outputs (i.e., labels), the mean absolute error is computed following Formula 3. Smaller is better.

$$MAE(y, \hat{y}) = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{n} \quad (3)$$

The R^2 score is a measure of how well future samples are likely to be predicted by the model, which is computed following Formula 4, where $\overline{y} = \frac{\sum_{i=1}^{n} y_i}{n}$. Larger is better.

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \overline{y}_i)^2} \quad (4)$$

4.5 Process

For compilers in the dataset [5], we use the same 100,000 test programs as the existing work [5], which serve as the new test programs to be prioritized.

First, we apply RO to the test programs and feed the prioritized test programs to the two compiler testing techniques, respectively. During the process, we record the execution time of each test program and which test programs trigger which bugs, then calculate the time spent on detecting each bug. The results of RO demonstrate the compiler testing results without using any accelerating approaches. To reduce the influence of random selection, we apply RO 10 times and calculate the average results.

Next, we apply COP and LET to each compiler under test in the two application scenarios by using the two compiler testing techniques, respectively. During this process, we also calculate the time spent on detecting each bug.

When evaluating the effectiveness of COP on the new dataset of 12 latest release versions of GCC and LLVM, we follow the similar process with the above, but we use different test programs since there is no test-program data of the new dataset. Here we use the DOL technique as the one to accelerate.

Furthermore, to evaluate the accuracy of COP on predicting coverage, we use 10-fold cross validation to assess the effectiveness of COP based on 2,000 test programs in the training set.

More specifically, during each fold cross validation, we apply COP to 1,800 test programs to train a prediction model on a version of a compiler, and then use the learnt model to predict coverage of each of remaining 200 test programs on a version (i.e., the same version or a later version) of the same compiler. Based on the predicted coverage and real coverage, we calculate the explained variance score, mean absolute error, and R^2 score on each fold cross validation, and calculate the corresponding mean values of these metrics on 10-fold cross validation.

Similarly, we apply RG to predict coverage of each test program. Note that in order to compare RG and COP, for RG we also split 2,000 test programs into 10 folds as the same as 10-fold cross validation for COP, and then calculate the values of metrics correspondingly. To reduce the influence of randomness, we apply RG 10 times and calculate the average results.

5 Results and Analysis

5.1 Acceleration Effectiveness

Table 2 shows the acceleration results of COP and LET in terms of time spent on detecting bugs for the dataset [5][7]. In this table, Column "Scenarios" presents the application scenarios, and the subject before an arrow is used to train a prediction model and the subject after an arrow is used to be accelerated; Column "Bug" presents the number of detected bugs; Column "RO" presents the average time spent on detecting the corresponding number of bugs without any acceleration approaches; Columns "ΔLET" and "ΔCOP" present the difference of the time spent on detecting the corresponding number of bugs between using LET/COP and using RO. **If the difference is less than zero, the corresponding approach accelerates compiler testing** because the used accelerating approach (LET or COP) spends less time than RO on detecting the corresponding number of bugs.

5.1.1 Overall Effectiveness

Table 2 shows that almost all values (109 in 116) of "ΔCOP" are smaller than zero, demonstrating that COP accelerates compiler testing in 93.97% cases. With the number of detected bugs increasing, the absolute values becomes much larger. That means that acceleration effectiveness of COP becomes more obvious as the testing proceeds. For the small number of values that are larger than zero, they are far smaller than the absolute values that are smaller than zero. For example, the largest value of those that are larger than zero is only 1.09, but the largest absolute value of those that are smaller than zero is 58.06. This also reflects the stably great effectiveness of COP on accelerating compiler testing. In addition, we further analyze the reason why COP has poor performance for some cases. Most cases of poor performance are caused by the higher setup time of COP. Since there are extra costs for COP (e.g., costs of predicting coverage) in prioritizing test programs, the first few bugs may be delayed. As time goes by, COP eventually shows better acceleration.

We further analyze acceleration effectiveness of COP by its speedup distribution. Figure 2 shows the speedup distribution of acceleration approaches[8], where the left one shows the speedup distribution of COP. The violin plots show the density of speedups at different values, and the box plots show the median and interquartile ranges. From the speedup distribution of COP, we find that its speedups

7. We do not use the bugs of LLVM-2.7 detected by DOL, because the number of detected bug is only one, which does not have any statistical significance.

8. For each r and each setting, we calculate the speedup of COP using Formula 1. We then analyze these speedups together. Here we do not show the deceleration cases.

TABLE 2
Time spent on bug detection (* 10^4 seconds)

Scenarios	Bug	DOL			EMI		
		RO	ΔLET	ΔCOP	RO	ΔLET	ΔCOP
GCC-4.3.0 → GCC-4.4.3	1	0.02	0.02	0.10	0.29	0.64	-0.02
	2	0.32	-0.07	-0.13	0.78	1.78	0.92
	3	0.89	-0.56	-0.40	1.43	1.45	0.28
	4	1.90	-1.12	-1.36	2.98	0.58	-1.20
	5	2.36	-1.17	-1.23	5.13	0.67	-1.22
	6	2.93	-1.51	-0.73	9.35	-3.55	-1.96
	7	3.96	-2.48	-1.62	12.21	-1.24	-2.43
	8	4.82	-1.84	-1.86	14.66	-3.05	0.10
	9	6.31	-2.15	-2.65	19.35	-0.88	-2.10
	10	7.58	-3.07	-3.66	23.25	-3.90	-5.48
	11	9.67	-5.15	-5.38	27.26	-7.48	-9.48
	12	12.11	-6.73	-7.42	29.69	-9.91	-4.86
	13	14.73	-6.85	-9.37	36.02	-13.73	-11.14
	14	17.47	-6.47	-11.58	40.54	-18.24	-15.32
	15	18.92	-4.34	-12.94	48.86	-16.01	-23.64
	16	21.08	-5.62	-14.34	53.90	-11.08	-28.68
	17	24.37	-4.31	-14.14	58.08	-11.64	-32.86
	18	26.39	-4.37	-14.05	61.69	-15.26	-24.47
	19	29.63	-4.99	-16.77	76.52	-23.99	-37.60
	20	32.08	-6.77	-16.77	78.96	-21.61	-37.51
	21	–	–	–	88.59	-28.98	-46.49
	22	–	–	–	107.05	-34.01	-54.00
	23	–	–	–	115.04	-35.63	-53.11
GCC-4.4.3 → GCC-4.4.3	1	0.02	0.02	0.08	0.29	0.64	-0.08
	2	0.32	-0.07	-0.19	0.78	1.78	-0.47
	3	0.89	-0.56	-0.75	1.43	1.45	-0.83
	4	1.90	-1.12	-1.46	2.98	0.58	-1.38
	5	2.36	-1.17	-1.44	5.13	0.67	0.48
	6	2.93	-1.51	-1.26	9.35	-3.55	-3.74
	7	3.96	-2.48	-1.93	12.21	-1.24	-6.60
	8	4.82	-1.84	-2.31	14.66	-3.05	-9.05
	9	6.31	-2.15	-3.43	19.35	-0.88	-13.52
	10	7.58	-3.07	-4.70	23.25	-3.90	-16.35
	11	9.67	-5.15	-5.51	27.26	-7.48	-17.39
	12	12.11	-6.73	-6.99	29.69	-9.91	-19.68
	13	14.73	-6.85	-9.61	36.02	-13.73	-20.98
	14	17.47	-6.47	-9.27	40.54	-18.24	-19.79
	15	18.92	-4.34	-7.85	48.86	-16.01	-26.25
	16	21.08	-5.62	-9.27	53.90	-11.08	-20.66
	17	24.37	-4.31	-11.54	58.08	-11.64	-17.75
	18	26.39	-4.37	-11.34	61.69	-15.26	-21.28
	19	29.63	-4.99	-11.79	76.52	-23.99	-36.10
	20	32.08	-6.77	-10.13	78.96	-21.61	-35.22
	21	–	–	–	88.59	-28.98	-37.87
	22	–	–	–	107.05	-34.01	-39.33
	23	–	–	–	115.04	-35.63	-44.09
LLVM-2.6 → LLVM-2.6	1	0.35	-0.06	-0.04	3.32	-0.29	-1.61
	2	1.06	-0.73	-0.32	8.56	-3.76	-6.12
	3	2.60	-1.54	-1.63	16.08	-11.28	-13.64
	4	4.33	-2.87	-3.29	23.25	-3.97	-20.81
	5	5.23	-1.63	-3.27	33.23	-13.95	-29.48
	6	6.83	-0.01	-3.99	47.86	-23.19	-39.85
	7	8.73	1.12	-2.83	56.30	-29.80	-47.42
	8	9.97	3.11	1.09	66.83	-27.50	-30.53
	9	13.37	-0.06	-0.94	86.07	-38.23	-49.77
	10	16.33	-0.53	-3.13	94.70	-43.85	-35.05
	11	20.25	-0.84	-5.46	–	–	–
	12	22.87	-2.85	-6.73	–	–	–
	13	25.45	-5.07	-2.19	–	–	–
	14	30.76	-9.60	-6.10	–	–	–
LLVM-2.7 → LLVM-2.7	1	–	–	–	3.19	-0.38	-1.06
	2	–	–	–	22.95	-20.14	-19.05
	3	–	–	–	61.84	-23.14	-46.50
LLVM-2.6 → LLVM-2.7	1	–	–	–	3.19	-0.38	-2.55
	2	–	–	–	22.95	-20.14	-22.28
	3	–	–	–	61.84	-23.14	-58.06

are ranging from 7.07% to 97.06%. In particular, the density of 50% speedup for COP is the largest, and the speedups of COP are over 50% in about half of cases. Therefore, COP accelerates compiler testing to a large degree.

Besides, we further analyze the acceleration effectiveness of COP on different compiler testing techniques and different application scenarios, whose results are shown in Table 3. In this table, Row "Acc.(%)" represents the percentage of cases where COP accelerates compiler testing; Row "Mean(%)" represents the mean speedup of COP in each

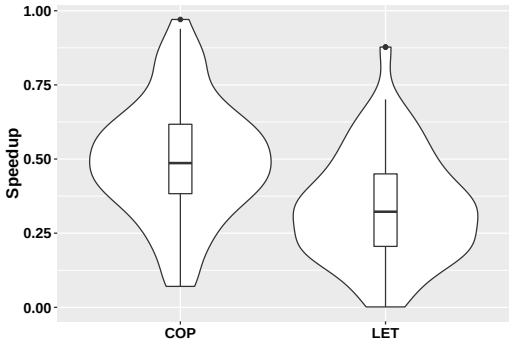


Fig. 2. Speedup distribution of acceleration approaches, COP and LET

TABLE 3
Effectiveness on different compiler testing techniques and different application scenarios

Summary	Techniques		Scenarios	
	DOL	EMI	Same-version	Cross-version
Acc.(%)	94.44	93.55	92.86	95.65
Mean(%)	46.12	54.90	47.60	55.78
p-value	0.000(✚)	0.000(✚)	0.000(✚)	0.000(✚)

setting; Row "p-value" represents the p-value of a paired sample Wilcoxon signed-rank test (at the significance level 0.05) in each setting, which reflects the significance in statistics. The p-values with (✚) represent that COP significantly accelerates compiler testing in corresponding settings. From this table, we find that COP accelerates compiler testing in almost all cases, more than 92.86%, no matter which compiler testing techniques and application scenario are applied. Furthermore, COP significantly accelerates compiler testing for both DOL and EMI on same-version and cross-version scenarios, whose mean speedups are ranging from 46.12% to 55.78%. Overall, COP always achieves great effectiveness for different compiler testing techniques on different application scenarios.

5.1.2 Comparison with LET

From Table 2, there are 14 cases where LET decelerates compiler testing while there are only 7 cases where COP decelerates compiler testing, demonstrating that COP has more stable acceleration effectiveness than LET. Moreover, we also find that the absolute values of Column "ΔCOP" are larger than those of Column "ΔLET" in most cases. That means that COP achieves better acceleration effectiveness than LET.

Similarly, we also analyze the speedup distribution of LET, which is also shown in Figure 2, the right one. From this figure, we find that the speedups of LET are ranging 0.14% to 87.75%. Both the minimum and maximum speedups are smaller than those of COP. Moreover, the density of about 30% speedup for LET is the largest, and the speedups of LET are over 50% in only 18.63% cases. That demonstrates that COP outperforms LET on accelerating compiler testing.

To further learn whether COP significantly outperforms LET, we perform a paired sample Wilcoxon signed-rank test (at the significance level 0.05), whose results are shown in Table 4. In this table, Row "p-value" represent the p-values

TABLE 4
Statistical analysis between COP and LET

Scenarios		GCC-4.4.3 → GCC-4.4.3	GCC-4.3.0 → GCC-4.4.3	LLVM-2.6 → LLVM-2.6
DOL	Mean(%)	17.61	20.26	14.26
	p-value	0.006(✚)	0.001(✚)	0.064
EMI	Mean(%)	22.45	43.81	55.27
	p-value	0.014(✚)	0.000(✚)	0.022(✚)

of the respective scenarios. The p-values with (✚) mean that COP significantly outperforms LET, and those without (✚) mean that there are no significant difference between COP and LET. Row "Mean (%)" represents the average speedups between COP and LET, which are computed by adapting Formula 1. More specifically, $TRO(r)$ in the adapted formula refers to the time spent on detecting r bugs through LET and $ACC(r)$ refers to the time spent on detecting r bugs through COP. Since the number of detected bugs by EMI on LLVM-2.7 is only three, the small number cannot be used to perform the paired sample Wilcoxon signed-rank test, and thus we do not show the results of accelerating testing of LLVM-2.7 in this table. In particular, the mean speedup of "LLVM-2.7→LLVM-2.7" for EMI is 25.28% and the mean speedup of "LLVM-2.6→LLVM-2.7" for EMI is 82.51%. From Table 4, we find that in most cases (5 in 6) COP significantly outperforms LET. Moreover, the mean improvements of COP over LET are ranging 14.26% to 55.27% in this table. Therefore, COP does perform significantly better than LET, the state-of-the-art acceleration approach.

Also, there are some cases where COP performs worse than LET. We further investigate the reason behind this phenomenon. On the one hand, most of these cases are also caused by the higher setup time of COP, since COP also has more extra costs (e.g., costs of predicting coverage) than LET. On the other hand, for other cases, this is because the features used in COP are unrelated to these bugs, which causes to delay the detection of these bugs. This is as expected, it is impossible to identify all features related to all bugs, and any prioritization approach cannot guarantee to accelerate testing for all bugs. That is, our approach aims to accelerate compiler testing in general, not for some specific bugs. A similar case is compiler optimization. A compiler optimization is designed to optimize compiling and execution of a program in general, and thus sometimes a compiler optimization may behave the opposite. In particular, it has been demonstrated that COP does significantly accelerate compiler testing in terms of overall effectiveness. In the future, we will consider adding more features for further improving our approach.

5.1.3 Acceleration Effectiveness on Latest Release Versions of GCC and LLVM

We further investigate the acceleration effectiveness of COP on the dataset of 12 latest release versions of GCC and LLVM using the DOL technique and the same-version scenario, whose results are shown in Table 5. In this table, Columns 3-5 present the time spent on bug detection of RO, LET, and COP; the last two columns present the mean speedup of COP compared with RO and LET respectively. For the 12 latest release versions, we detect 19 compiler bugs in total by running the generated test programs[9]. From Table 5, overall, COP accelerates the testing of the latest release versions in 84.21% cases. The mean speedup of COP is 68.74%, outperforming the mean speedup on old versions in the dataset [5] (i.e., 51.01%). That is, COP still significantly accelerates compiler testing for latest release versions. Furthermore, COP performs better than LET in 73.68% cases on the latest release versions, and the mean improvement of COP over LET is 49.18%. That is, COP still outperforms LET on latest release versions of GCC and LLVM.

TABLE 5
Effectiveness of COP on latest release versions including time spent on bug detection (* 10^4 seconds) and overall speedups

Subjects	Bugs	RO	ΔLET	ΔCOP	$\uparrow_{RO}$	$\uparrow_{LET}$
GCC-6.1.0	1	24.93	-2.73	-19.02		
	2	83.26	-24.10	-48.45		
GCC-6.1.0	1	13.39	-3.94	-3.97		
	2	52.29	26.76	25.47		
	3	70.36	13.51	15.44		
GCC-6.3.0	1	48.78	-39.13	-46.19		
GCC-6.4.0	1	39.97	-17.05	15.30		
	2	110.25	-24.87	-51.81		
GCC-7.1.0	1	25.47	-22.43	-23.44		
	2	80.23	-24.79	-33.9	68.74%	49.18%
GCC-7.2.0	1	22.26	-14.80	-2.15		
	2	65.97	20.36	-31.81		
GCC-7.3.0	1	61.54	-26.37	-46.76		
LLVM-4.0.0	1	57.39	-42.64	-16.25		
LLVM-4.0.1	1	46.74	-26.08	-41.55		
LLVM-5.0.0	1	21.20	3.21	-14.31		
	2	62.23	-27.31	-3.73		
LLVM-5.0.1	1	34.77	-3.40	-32.82		
LLVM-6.0.0	1	49.90	0.46	-49.54		

5.1.4 Acceleration Effectiveness on Swarm Testing

In compiler testing, there are various test program generation methods. It is interesting to investigate whether COP is able to accelerate compiler testing based on the test programs generated by different methods. Here we used a state-of-the-art method, swarm testing [28], as the representative test program generation method. More specifically, swarm testing generates test programs by tuning a set of flags of Csmith to enable/disable C language features, so as to detect more bugs during the given period of time [28].

In the study, we first used swarm testing to generate 30,000 test programs for GCC-4.4.3 and LLVM-2.6, respectively. Then, we applied COP to accelerate their testing in the same-version scenario. Table 6 shows the acceleration effectiveness on swarm testing. In this table, Column "RO_{sw}" presents the testing effectiveness using the test programs generated by swarm testing; Column "ΔLET_{sw}" and ΔCOP_{sw} present the acceleration results of LET and COP based on the test programs generated by swarm testing. From this table, using swarm testing, COP still does accelerate compiler testing in 82.35% cases except the first few bugs. Compared with swarm testing, the average speedup of COP is 38.37%, and compared with LET based on swarm testing, the average speedup of COP is 14.88%. These results are similar to those using the test programs generated by vanilla Csmith. Therefore, the results demonstrate that

9. All the bug-triggering test programs for the 19 bugs detected in the latest release versions in the study can be found at https://github.com/JunjieChen/COPprograms.

TABLE 6
Effectiveness of COP with swarm testing (including time spent on bug detection (* 10^4 seconds) and overall speedups)

Subjects	Bugs	RO_{sw}	ΔLET_{sw}	ΔCOP_{sw}	$\uparrow_{RO_{sw}}$	$\uparrow_{LET_{sw}}$
GCC-4.4.3	1	0.12	-0.07	-0.09	38.37%	14.88%
	2	0.24	-0.10	0.03		
	3	0.58	-0.18	-0.22		
	4	1.01	-0.61	-0.53		
	5	4.08	-2.48	-3.07		
	6	4.21	-0.99	-3.11		
	7	4.27	-0.99	-2.93		
	8	4.59	-0.98	-3.21		
	9	4.66	-0.82	-2.13		
	10	5.12	-1.12	-2.07		
	11	7.55	-3.53	-3.88		
	12	9.22	-0.41	-0.17		
LLVM-2.6	1	0.01	0.00	0.04		
	2	0.06	0.02	0.07		
	3	0.51	0.00	-0.12		
	4	1.08	-0.09	-0.41		
	5	2.71	-1.29	-1.83		

TABLE 7
Effectiveness of COP for predicting coverage in the same-version application scenario

Subjects		EV		MAE		R^2	
		COP	RG	COP	RG	COP	RG
GCC	4.3.0→4.3.0	0.899	-1.423	0.039	0.309	0.840	-1.497
	4.4.0→4.4.0	0.884	-1.672	0.036	0.302	0.859	-1.733
	4.5.0→4.5.0	0.897	-1.600	0.034	0.303	0.874	-1.634
	4.6.0→4.6.0	0.891	-1.686	0.034	0.300	0.869	-1.722
LLVM	2.6→2.6	0.799	-1.134	0.051	0.325	0.782	-1.162
	2.7→2.7	0.790	-1.179	0.050	0.322	0.774	-1.202
	2.8→2.8	0.822	-1.172	0.049	0.322	0.807	-1.193

TABLE 8
Effectiveness of COP for predicting coverage in the cross-version application scenario

Subjects		EV		MAE		R^2	
		COP	RG	COP	RG	COP	RG
GCC	4.3.0→4.4.0	0.761	-1.529	0.067	0.306	0.747	-1.571
	4.4.0→4.5.0	0.749	-1.708	0.073	0.300	0.706	-1.749
	4.5.0→4.6.0	0.874	-1.786	0.052	0.297	0.854	-1.825
LLVM	2.6→2.7	0.476	-1.092	0.116	0.327	0.469	-1.116
	2.7→2.8	0.535	-1.283	0.100	0.316	0.502	-1.317

COP is able to stably accelerate compiler testing no matter which test program generation method (swarming testing or vanilla Csmith) is used. This further reflects that COP is orthogonal to the test program generation methods.

5.2 Prediction Effectiveness

Table 7 shows the effectiveness of COP on predicting coverage in the same-version application scenario. From this table, we find that COP significantly outperforms RG in terms of all used metrics. In particular, the EV and R^2 values of RG are negative, which means that it is unpredictable using RG. The MAE values of COP are extremely small (i.e., 0.034∼0.051), demonstrating its high accuracy on predicting coverage in the same-version application scenario. Moreover, both EV and R^2 of COP are large, demonstrating its high predictability of coverage in the same-version application scenario.

Table 8 shows the effectiveness of COP on predicting coverage in the cross-version application scenario. From this table, we also find that COP significantly outperforms RG no matter which metric is used. The MAE values of COP are also small (i.e., 0.052∼0.116), demonstrating its high accuracy on predicting coverage in the cross-version application scenario. We also find that the MAE values in the cross-version application scenario are larger than those in the same-version application scenario. This is reasonable, because there are evolutions between versions, which incur inaccuracies in prediction. Similarly, the EV and R^2 values in the cross-version application scenario are smaller than those in the same-version application scenario. Moreover, we find that the prediction results for LLVM in the cross-version application scenario become worse than those for GCC. The reason is that the changes between versions of LLVM tend to be larger than those of GCC.

In general, COP indeed effectively predicts coverage no matter which application scenario is applied. The smaller the changes between the training version and the predicting version are, the more accurate the prediction is. That is, when we train a prediction model based on a version, if its later versions have small changes with it, we can use it to predict coverage all the time in order to save the off-line training costs. As the first attempt to predict coverage statically for compilers, it shows a promising direction.

6 Threats to Validity

•Threats to Internal Validity

The threats to internal validity mainly lie in the implementations of COP and EMI. To reduce these threats, the first two authors review all the code. Besides, we adopt the same tools as the existing work [3] to implement EMI, including LibTooling Library of Clang[10] and Gcov.

•Threats to External Validity

The threats to external validity mainly lie in the subjects and test programs.

To reduce the threat from subjects, we use the same dataset as the existing work [5], and also construct a new dataset that includes 12 latest release versions of GCC and LLVM to evaluate the acceleration effectiveness of COP. Also, to evaluate the accuracy of COP on predicting coverage, we additional use more release versions based on the dataset [5]. In the future, we will use more compilers and more versions as subjects.

To reduce the threat from test programs, we use test programs randomly generated by Csmith same as the prior work [1], [2], [3], [4], [5]. However, these test programs may be not necessarily representative of C programs generated by other tools. Nevertheless, the results obtained from Csmith is practice-relevant. When testing C compilers, Csmith is the state-of-the-art tool to generate C programs and is the only tool used in recent compiler testing research [3], [28]. Test generators of other compilers are often built upon Csmith, such as CLsmith for OpenCL compilers [29]. That is, the evaluation results on Csmith best reflects the performance of the tool in practice.

The number of test programs may also impact the evaluation of COP effectiveness. Actually, COP is independent of the number of generated test programs. COP accelerates compiler testing by distinguishing different test programs with different test capabilities. It will predict the coverage

10. http://clang.llvm.org/docs/LibTooling.html.

of each test program and schedule their execution order in order to improve test efficiency. Therefore, COP can help accelerate compiler testing when the number of test programs is low (the test coverage is subject to change) and when the number of test programs is high (the test coverage reaches a stable state). In particular, we conduct a small experiment to check the coverage state including statement coverage, branch coverage, and function coverage of the 100,000 test programs used in our study. We find that all the three kinds of coverage are nearly unchanged with the number of test programs increasing after about 10,000. That is, the used test programs in our study actually have reached a stable coverage state. Therefore, our experimental results have demonstrated that COP indeed works when reaching a stable state.

•Threats to Construct Validity

The threats to construct validity mainly lie in how the results are measured, the setting of ψ, and the used optimization level for coverage collection when building the prediction model.

When measuring acceleration effectiveness, we use Correcting Commits to automatically identify duplicated bugs [1]. As Correcting Commits depends on developers edition, different bugs may be regarded as the same one. However, it may not be a big threat since developers do not tend to fix many bugs in a commit to guarantee the quality of compilers [1].

For the setting of ψ, to investigate its impact, we further conduct an experiment to test different thresholds using GCC-4.4.3. More specifically, we set ψ as the value (i.e., the bug-revealing probability per unit time of the test program) at the 1/3, 1/2, and last positions in the ranking list of test programs respectively. Figure 3 shows the effectiveness of COP using different values of ψ. Overall, the current setting of ψ (i.e., the value at the 2/3 position) performs the best among them, since it usually spends the least time to find bugs shown in this figure. That demonstrates that the current setting seems to be the best choice for COP. We further analyze the reason why other settings perform worse. We find that, for the value at the last position, many test programs with extremely small bug-revealing probabilities per unit time are ranked at earlier positions through clustering. For the values at the 1/2 and 1/3 positions, many test programs that are able to trigger bugs are ranked at later positions since their bug-revealing probabilities per unit time are smaller than the set threshold. That is, there actually exists a tradeoff, and the value at the 2/3 position is a good balance.

For the used compiler optimization level for coverage collection in COP, we used the highest optimization level "-O3" in our study. Compiler optimization levels can affect the coverage, thus they may also affect the effectiveness of our approach COP. To investigate the impact, we further conduct an experiment to test different optimization levels including "-O1" and "-O2" using GCC-4.4.3. The results are shown in Table 9. From this table, COP always significantly accelerates compiler testing and significantly outperforms LET no matter which optimization level is used for coverage collection. For example, compared with RO, the average speedups of COP using "-O1", "-O2", and "-O3" are 54.67%, 52.51%, and 48.36%, respectively. Therefore, COP can achieve stable and good results using different optimization levels for coverage collection.

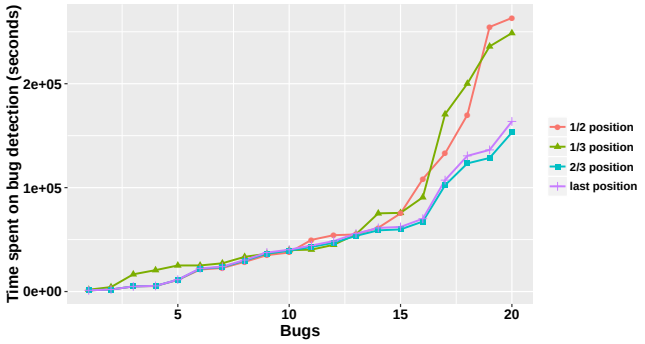


Fig. 3. Impact of ψ for COP on accelerating compiler testing

TABLE 9
Impact of different optimization levels on COP

Level		-O1	-O2	-O3
COP v.s. RO	Mean(%)	54.67	52.51	48.36
	p-value	0.000(✚)	0.000(✚)	0.000(✚)
COP v.s. LET	Mean(%)	27.68	24.23	17.61
	p-value	0.000(✚)	0.000(✚)	0.006(✚)

7 Discussion

7.1 Why COP Works

Our experimental results have demonstrated the accuracy of COP on predicting coverage on compilers, but it still has deviations. Even though it utilizes the predicted coverage information that has some deviations, COP still achieves significant accelerating effectiveness for compiler testing. On the one hand, it shows the potential of COP; on the other hand, it is also interesting for us to know the reason behind it. In this section, we further investigate the reason. In particular, we use GCC-4.3.0 and LLVM-2.6 as the representatives to analyze the prediction accuracy of each predicted file. Table 10 and Table 11 present the top-10 files that are predicted most accurately by COP on GCC-4.3.0 and LLVM-2.6, respectively. Here "the most accurately" refers to the smallest values of MAE. In this table, the last three columns represent the names of files, the values of mean absolute error, and the description of their functionalities, respectively. From the two tables, we find that most files in the top-10 files are quite important and error-prone for compilers. For example, for GCC in Table 10, four of ten are tree-optimization-related files, and we can find that a large number of GCC bugs occurring at tree optimization parts from GCC Bugzilla[11]. Similarly, for LLVM in Table 11, "X86ISelLowering.cpp" and "SemaExprCXX.cpp" are reported as the two of top-10 files containing the largest number of bugs [30]. The highly accurate prediction for such files facilitates to separate test programs triggering more different bugs into different groups, so that COP accelerates compiler testing to a larger extent. Therefore, the reason why COP achieves great acceleration effectiveness based on the imperfect predicted coverage information may be that, COP

11. https://gcc.gnu.org/bugzilla/.

predicts much more accurate results for the more important and error-prone files.

7.2 Extension of COP

For COP, there are some possible extensions to improve its effectiveness. First, the key insight of COP to be able to predict coverage is that test programs with some specific features are more likely to cover some specific code regions of compilers. Currently, COP identifies three categories of features as Section 3.1.1 presented, including language features, operation features, and structure features. In fact, these features are not all features that can reflect compiler coverage, and it is also impossible for us to manually identify all these features. Deep learning provides an opportunity to automatically identify features from test programs, one of whose preconditions is requiring big data. Fortunately, in compiler testing, it is easy to acquire a huge number of test programs by some test-generation tools like Csmith. Therefore, in the future, we will use deep learning to automatically augment features in COP.

Second, when applying COP in the cross-version application scenario, it has the label alignment problem due to the existence of newly added files in the later version and out-of-date files in the prior version. Here we identify such files by matching file names between versions. Currently, COP removes these newly added files and out-of-date files from the label set. That is, COP does not consider these files when distinguishing test programs with different test capabilities. In fact, some files may be not real newly added files or out-of-date files, which may be forged by renaming file names, merging multiple files into a single file, or separating a file into multiple files, etc. Therefore, we will try to identify such forgeries and better match these files between versions, in order to better deal with the label alignment problem in COP.

Third, after clustering, COP prioritizes test programs by enumerating each group to select the test program with the largest bug-revealing probability per unit time in each group. In the future, we will further improve the prioritization strategy. For example, we can try to adopt the roulette wheel selection strategy, which has been widely used in the genetic algorithm [31]. Instead of enumerating each group to select test programs, it assigns a probability to each group, and the group with the larger probability is more likely to be selected in each selection.

7.3 Applications of COP

COP is not specific to the compiler testing techniques used in our study, i.e., DOL and EMI. It can be applied to accelerate any compiler testing techniques, because it is orthogonal with these compiler testing techniques. For example, RDT is also a widely-used compiler testing technique, which detects compiler bugs by comparing results produced by different compilers. To accelerate RDT, we first apply COP to prioritize test programs, and then run each test program using RDT based on the prioritization results.

Our evaluation has demonstrated the great effectiveness of COP on accelerating C compiler testing. Actually, COP can be generalized to accelerate testing of compilers of other programming languages (e.g., Java). COP utilizes three general categories of features: language features, operation features, and structure features. These features are common across all programming languages. When applying COP to other programming languages, for the first category (i.e., language features), we can extract them according to the constructs of the programming languages. For the other two categories (those features reused from Csmith in C), we can actually obtain them by analyzing historical compiler bugs (with the corresponding test programs triggering the bugs). More specifically, we investigate which operations among variables and structure properties (e.g., nested loops) in the test programs caused the corresponding bugs. In the future, we will try to apply COP to more compilers of other programming languages.

Besides, COP may be also generalized to accelerate testing of other software systems, e.g., operator systems, browsers, and static analyzers. Similar to compilers, the test inputs of these software systems are test programs, which provides an opportunity for COP to identify features related to coverage prediction. That is, COP can be also applied to predict coverage and then prioritize test programs based on the predicted coverage information, so as to accelerate their testing process.

One of the most important parts in COP is coverage prediction. Our work uses the predicted coverage information to accelerate compiler testing. In fact, due to many drawbacks of collecting coverage dynamically for compilers, a lot of software testing processes are limited, e.g., test-suite reduction, test-program generation, and test-suite construction. However, our coverage prediction method makes them feasible to a large degree. That is, there are actually a lot of applications of our coverage prediction method for compilers. For example, we can utilize the predicted coverage information to guide the construction of a test suite. When constructing a test suite, we need to measure the adequacy of the test suite. Based on the existing work [9], test coverage is one of the most widely-used measurements, but dynamically collecting coverage for a test suite is extremely time-consuming for compilers. Therefore, statically predicting test coverage is helpful. As the first attempt to predict coverage statically, our coverage prediction method provides a light-weight and accurate way to acquire module-level coverage (i.e., file-level coverage), which is likely to help guide the construction of a test suite. Finer-grained coverage (e.g., statement coverage) may be better for test-suite construction, and thus it would be interesting to explore if finer-grained coverage can be predicted and if it can be used to construct a more effective test suite. This is a future work to be explored.

8 Related Work

8.1 Compiler Testing

Compiler testing is a difficult and essential task due to the complexity and importance of compilers, and thus it has attracted extensive attentions from researchers [32], [33], [34], [35], [36]. Research on compiler testing can be divided into three main aspects, i.e., test-program generation, test-oracle construction, and test-execution optimization.

Test programs are the inputs of compilers, which must strictly meet complex specifications (e.g., the C99 specifica-

TABLE 10
Top-10 files that are predicted most accurately by COP on GCC-4.3.0

Top	Name	MAE	Description
1	prefix.c	0.005	Updating a path, both to canonicalize the directory format and to handle any prefix translation.
2	tree-ssa-ccp.c	0.006	Conducting conditional constant propagation based on the SSA propagation engine.
3	tree-sra.c	0.008	Replacing a non-addressable aggregate with a set of independent variables.
4	tree-vectorizer.c	0.008	Vectorizing loops.
5	ipa.c	0.008	Marking visibility of all functions.
6	gimple-low.c	0.009	Lowering GIMPLE into unstructured form.
7	tree-ssa-copy.c	0.009	Performing const/copy propagation and simple expression replacement.
8	gt-passes.h	0.010	Type information for passes.c.
9	builtins.c	0.010	Dealing with type-related information.
10	double-int.c	0.012	Operations with long integers.

TABLE 11
Top-10 files that are predicted most accurately by COP on LLVM-2.6

Top	Name	MAE	Description
1	DelaySlotFiller.cpp	0.001	Returning a pass that fills in delay slots in Sparc MachineFunctions.
2	Thumb1InstrInfo.h	0.002	Processing instruction information.
3	UnifyFunctionExitNodes.h	0.003	Ensuring that functions have at most one return and one unwind instruction in them.
4	SymbolTableListTraitsImpl.h	0.003	Instantiating explicitly where needed to avoid defining all this code in a widely used header.
5	IdentifierResolver.h	0.004	Using for lexical scoped lookup, based on declaration names.
6	MachineInstr.cpp	0.005	Methods common to all machine instructions.
7	SPUSubtarget.h	0.005	Declaring the Cell SPU-specific subclass of TargetSubtarget.
8	ParsePragma.h	0.005	Defining #pragma handlers for language specific pragmas.
9	X86ISelLowering.cpp	0.005	Defining the interfaces that X86 uses to lower LLVM code into a selection DAG.
10	SemaExprCXX.cpp	0.005	Implementing semantic analysis for C++ expressions.

tion for C programs). Due to its basic role in compiler testing, a lot of work focuses on test-program generation [37], [38]. In particular, Boujarwah and Saleh presented a survey to introduce test-program generation methods proposed before 1997 [39]. In general, the methods to generate test programs can be divided into two categories. The first one is to generate totally new test programs, while the second one is to generate test programs based on existing test programs. Most test-program generation methods in the first category have been introduced by the survey [39]. Besides, Yang et al. [2], [40], [41] proposed and implemented a tool, called Csmith, to randomly generate C programs for testing C compilers, which is one of the most widely-used methods so far. Also, Lidbury et al. [29] developed a tool called CLsmith, based on Csmith, to generate test programs for OpenCL compilers. Nagai et al. [42], [43] proposed to generate test programs, which contain randomly generated arithmetic expressions without undefined behaviors, to test C compilers' arithmetic optimization. Lindig [44] proposed to randomly generate C programs in order to test the consistency of C compilers, and implemented a tool (Quest), which is type-directed and can be controlled by the user, rather than generates programs based on control flow and arithmetic. Pałka et al. [45] proposed to randomly generate lambda terms in order to test an optimizing compiler, which mainly addresses the issue of type correctness in the generation. Alipour et al. [46] proposed directed swarm testing to generate test programs focusing on only part of a compiler through statistical data analysis on past testing results. Zhao et al. [47] developed an integrated tool (JTT), which automatically generates programs in order to test UniPhier, an embedded C++ compiler. For the second category of generating test programs, Le et al. [3] proposed to generate test programs by mutating existing test programs. More specifically, they generate equivalent variants with the existing ones by randomly removing unexecuted code regions from the existing test programs. Also, Le et al. [15], Sun et al. [16], and Zhang et al. [48] further proposed more mutation strategies to generate test programs based on existing test programs. Following the existing work [1], [2], [4], [5], [15], [23], our work uses the test programs generated by Csmith to test compilers. Please note that our approach COP aims to accelerate compiler testing by prioritizing the original test programs generated by Csmith rather than generate any new test programs. Therefore, if the original test programs cannot find any bugs for a compiler, COP cannot find bugs either. In particular, the most relevant work to ours in test-program generation is swarm testing [28], which generates test programs by tuning a set of flags of Csmith to enable/disable C language features, so as to detect more bugs during the given period of time. Actually, COP is orthogonal to swarm testing. Swarm testing tries to generate more diverse test programs, while COP predicts the test capabilities of generated test programs and then prioritizes these generated test programs. That is, COP can further accelerate swarm testing by taking the generated test programs through swarm testing as inputs.

Test-oracle construction problem is a long-term challenge in compiler testing, since developers can hardly determine whether the actual output of the compiler under test as expected given a test program. To address the test-oracle construction problem, McKeeman et al. [13] coined the term of differential testing, which is a form of random testing. In particular, differential testing needs two or more comparable compilers and determines whether some compilers have bugs by comparing the results produced by these compilers. Based on differential testing, Le et al. [23] proposed to detect the bugs of link-time optimizers in compilers by random-

ized stress-testing. Also, Sun et al. [14] proposed to find compiler warning defects by randomized differential testing. Besides, Le et al. [3] proposed equivalence modulo inputs (EMI) to address the test-oracle construction problem. EMI determines whether the compiler under test contains bugs by comparing the results produced by the original test program and its equivalent variants under given test inputs. In particular, EMI has three instantiations, Orion [3], Athena [15], and Hermes [16]. Tao et al. [49] proposed to test compilers by constructing metamorphic relations, e.g., the equivalent relation. Donaldson et al. [35], [50] utilized metamorphic testing to detect the bugs of graphics compilers by designing a set of semantics-preserving transformations. Recently, Chen et al. [1] conducted an empirical study to compare the strength of different test oracles, including randomized differential testing (RDT), a variant of RDT called different optimization levels (DOL), and EMI. Our work considers two methods of test-oracle construction, including differential-testing based test-oracle construction (i.e., DOL) and EMI-based test-oracle construction (i.e., Orion).

Test-execution optimization refers to optimize the execution of test programs so as to improve compiler testing, e.g., test-suite reduction and test-program prioritization. Woo et al. [51] proposed to remove redundant test programs from a test suite by measuring the redundancy of test programs from the viewpoint of the intermediate representation. Chae et al. [52] proposed to utilize intermediate-code coverage to reduce a test suite for testing retargeted C compilers for embedded systems. Furthermore, Chen et al. [4] proposed a text-vector based test-program prioritization approach to accelerating compiler testing. More specifically, their approach transforms test programs into a set of text-vectors, and prioritizes them by calculating the distance between each text-vector and the origin vector (0,0,...,0). Chen et al. [5] proposed to utilize historical bug information to predict bug-revealing probabilities per unit time of test programs, and then prioritize them based on the information to accelerate compiler testing. The proposed test-suite reduction approaches for compilers [51], [52] cannot be applied to accelerate compiler testing, because they need to dynamically collect coverage information, but our work accelerates compiler testing by statically predicting coverage information. In this category, our work actually belongs to the test-program prioritization, but different from the existing ones, our approach distinguishes whether different test programs with different test capabilities by predicting coverage statically, and then prioritizes test programs based on clustering results to accelerate compiler testing.

8.2 Test Prioritization

During the development of decades, there is a large amount of research on test prioritization. Following the existing work [53], research on test prioritization can be mainly classified into four categories. The first category focuses on the criteria used in test prioritization such as the widely-used structure code coverage criterion [54], [55], [56], [57]. Besides, Elbaum et al. [58] proposed to use the probability of exposing faults, Mei et al. [59] investigated dataflow coverage, and Korel et al. [60] used the coverage of system model instead of code coverage. The second category focuses on the algorithms used in test prioritization. Rothermel et al. [54] proposed greedy algorithms, i.e., the total and additional algorithms, to prioritize tests, which have become the most widely-used algorithms in test prioritization. Furthermore, researchers viewed test prioritization as a searching problem and thus proposed many meta-heuristics algorithms [61] to address this problem, e.g., the 2-optimal strategy, hill-climbing strategy, and genetic programming based strategy. Also, Jiang et al. [62] present the adaptive random strategy. The third category focuses on the constraints that affect test prioritization, e.g., time constraints [63]. Some work in this category also investigated the influence of the constraints, and proposed prioritization techniques specific to some constraints [64], [65], [66], [67], [68]. The fourth category focuses on the empirical studies about test prioritization [54], [63], [67], [69], [70], [71], [72], including evaluating the effectiveness of various test prioritization approaches, and investigating the influence of some factors in test prioritization.

Our work also prioritizes test programs in order to accelerate compiler testing. The differences between our work and the existing work on accelerating compiler testing have been discussed in Section 8.1. In the traditional test prioritization, the most related work is cluster-based test prioritization [73], [74], [75]. Yoo et al. [73] proposed to cluster tests based on the dynamic runtime behaviors of tests to reduce the cost of human-interactive prioritization. However, our approach does not belong to the human-interactive test prioritization. That is, the goal of using clustering in our work is different from that in their work. FU et al. [75] proposed a cluster-based test prioritization based on coverage information of tests in historical executions. More specifically, it clusters tests that have different properties in the different groups. Similarly, Carlson et al. [74] conducted a study to investigate whether clustering can improve test prioritization. It also clusters tests based on history information. Similar to them, our approach clusters test programs into groups and different groups are more likely to have different test capabilities, and then prioritizes test programs based on the clustering results. However, different from them, we cannot use coverage information in historical executions for compiler testing, and thus our work proposed the first method to predict coverage statically for compilers, which is also an innovation in our work. Also, during the prioritization in each cluster, we use the bug-revealing probability per unit time of each test program predicted by LET, which is also different from these work.

Besides, our work is also related to machine-learning based test prioritization, since our approach also uses machine learning. Spieker et al. [76] utilized reinforcement learning to select and prioritize tests based on their duration, previous last execution and failure history. Busjaeger and Xie [77] utilized machine learning to integrate multiple existing test prioritization approaches so as to apply test prioritization in industrial environments. Wang et al. [78] proposed quality-aware test prioritization, which leveraged code inspection techniques, i.e., a typical statistic defect prediction model and a typical static bug finder, to predict fault-proneness of source code, and then applied coverage-based test prioritization algorithms. Different from these work, our approach utilizes machine learning to predict

coverage information of new test programs for compilers, and then prioritizes them based on the predicted coverage information to accelerate compiler testing.

9 Conclusion

To accelerate compiler testing, some approaches have been proposed recently. These approaches prioritize test programs based on some criteria to execute test programs that are more likely to trigger compiler bugs earlier. However, they ignore an important aspect in compiler testing, where different test programs may have similar test capabilities. The neglect of this problem may largely discount the effectiveness of the existing acceleration approaches. Intuitively, if the coverage of test programs is very different, these test programs tend to have different test capabilities. However, it is infeasible to collect test coverage dynamically in compiler testing because test programs are generated on the fly. In this paper, we propose COP to predict test coverage statically for compilers, and then prioritize test programs by clustering them according to the predicted coverage information. Our experimental results confirm the acceleration effectiveness of COP, which achieves an average of 51.01% speedup in test execution time on the existing constructed dataset, achieves an average of 68.74% speedup on the new dataset of 12 latest release versions of GCC and LLVM, and even outperforms the state-of-the-art acceleration approach (i.e., LET) significantly by improving 17.16% ∼82.51% speedups in different settings on average.

References

[1] 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.

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

[3] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th Conference on Programming Language Design and Implementation*, 2014, p. 25.

[4] 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 *Proceedings of the 9th International Conference on Software Testing, Verification and Validation*, 2016, pp. 266–277.

[5] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing," in *ICSE*, 2017, to appear.

[6] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013, pp. 197–208.

[7] H. Zhu, P. A. Hall, and J. H. May, "Software unit test coverage and adequacy," *Acm computing surveys*, vol. 29, no. 4, pp. 366–427, 1997.

[8] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Transactions on software Engineering*, vol. 29, no. 3, pp. 195–209, 2003.

[9] A. S. Namin and J. H. Andrews, "The influence of size and coverage on test suite effectiveness," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009, pp. 57–68.

[10] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[11] G. Rothermel, M. J. Harrold, J. Von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Software Testing, Verification and Reliability*, vol. 12, no. 4, pp. 219–249, 2002.

[12] H. Zhong, L. Zhang, and H. Mei, "An experimental study of four typical test suite reduction techniques," *Information and Software Technology*, vol. 50, no. 6, pp. 534–546, 2008.

[13] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[14] C. Sun, V. Le, and Z. Su, "Finding and analyzing compiler warning defects," in *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[15] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *Proceedings of the 2015 International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015, pp. 386–399.

[16] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016, pp. 849–863.

[17] B. Jiang and W. K. Chan, "Input-based adaptive randomized test case prioritization: A local beam search approach," *Journal of Systems & Software*, vol. 105, no. C, pp. 91–106, 2015.

[18] R. Xu and D. Wunsch, "Survey of clustering algorithms," *IEEE Transactions on neural networks*, vol. 16, no. 3, pp. 645–678, 2005.

[19] D. Pelleg, A. W. Moore *et al.*, "X-means: Extending k-means with efficient estimation of the number of clusters," in *Proceedings of the 17th International Conference on Machine Learning*, 2000, pp. 727–734.

[20] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[21] J. Dai and Q. Xu, "Attribute selection based on information gain ratio in fuzzy rough set theory with application to tumor classification," *Applied Software Computing*, vol. 13, no. 1, pp. 211–221, 2013.

[22] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.

[23] V. Le, C. Sun, and Z. Su, "Randomized stress-testing of link-time optimizers," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 327–337.

[24] Hall, Mark, Frank, Eibe, Holmes, Geoffrey, Pfahringer, Bernhard, Reutemann, and Peter, "The weka data mining software: an update," *Acm Sigkdd Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[25] C. H. Achen, "What does "explained variance" explain?: Reply," *Political Analysis*, vol. 2, no. 1, pp. 173–184, 1990.

[26] C. J. Willmott and K. Matsuura, "Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance," *Climate Research*, vol. 30, no. 1, p. 79, 2005.

[27] R. G. Carpenter, "Principles and procedures of statistics with special reference to the biological sciences," *Annals of the New York Academy of Sciences*, vol. 682, no. 12, pp. 283–295, 1960.

[28] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 78–88.

[29] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proceedings of the 36th Conference on Programming Language Design and Implementation*, 2015, pp. 65–76.

[30] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in gcc and llvm," in *Proceedings of the 25th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2016, pp. 294–305.

[31] A. Lipowski and D. Lipowska, "Roulette-wheel selection via stochastic acceptance," *Physica A: Statistical Mechanics and its Applications*, vol. 391, no. 6, pp. 2193–2196, 2012.

[32] A. S. Kossatchev and M. A. Posypkin, "Survey of compiler testing methods," *Programming & Computer Software*, vol. 31, no. 1, pp. 10–19, 2005.

[33] M. Pflanzer, A. F. Donaldson, and A. Lascu, "Automatic test case reduction for opencl," in *Proceedings of the 4th International Workshop on OpenCL*, 2016, p. 1.

[34] M. Boussaa, O. Barais, B. Baudry, and G. Sunyé, "Notice: A framework for non-functional testing of compilers," in *Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability & Security*, 2016.

[35] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated testing of graphics shader compilers," *PACMPL*, vol. 1, no. OOPSLA, pp. 93:1–93:29, 2017.

[36] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," in *ACM Sigplan Con-*

ference on Programming Language Design and Implementation, 2016, pp. 85–99.

[37] K. V. Hanford, "Automatic generation of test cases," *IBM Systems Journal*, vol. 9, no. 4, pp. 242–257, 1970.

[38] R. L. Sauder, "A general test data generator for cobol," in *Proceedings of the 1962 spring joint computer conference*, 1962, pp. 317–323.

[39] A. S. Boujarwah and K. Saleh, "Compiler test case generation methods: a survey and assessment," *Information & Software Technology*, vol. 39, no. 9, pp. 617–625, 1997.

[40] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of the 34th Conference on Programming Language Design and Implementation*, vol. 48, no. 6, 2013, pp. 197–208.

[41] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proceedings of the 33rd Conference on Programming Language Design and Implementation*, vol. 47, no. 6, 2012, pp. 335–346.

[42] E. Nagai, H. Awazu, N. Ishiura, and N. Takeda, "Random testing of c compilers targeting arithmetic optimization," in *Workshop on Synthesis And System Integration of Mixed Information Technologies*, 2012, pp. 48–53.

[43] E. Nagai, A. Hashimoto, and N. Ishiura, "Scaling up size and number of expressions in random testing of arithmetic optimization of c compilers," in *Workshop on Synthesis And System Integration of Mixed Information Technologies*, 2013, pp. 88–93.

[44] C. Lindig, "Random testing of c calling conventions," in *Proceedings of the 6th international symposium on Automated analysis-driven debugging*, 2005, pp. 3–12.

[45] M. H. Pałka, K. Claessen, A. Russo, and J. Hughes, "Testing an optimising compiler by generating random lambda terms," in *Proceedings of the 6th International Workshop on Automation of Software Test*, 2011, pp. 91–97.

[46] M. A. Alipour, A. Groce, R. Gopinath, and A. Christi, "Generating focused random tests using directed swarm testing," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 70–81.

[47] C. Zhao, Y. Xue, Q. Tao, L. Guo, and Z. Wang, "Automated test program generation for an industrial optimizing compiler," in *2009 ICSE Workshop on Automation of Software Test*, 2009, pp. 36–43.

[48] Q. Zhang, C. Sun, and Z. Su, "Skeletal program enumeration for rigorous compiler testing," in *Proceedings of the 38th Conference on Programming Language Design and Implementation*, 2017, to appear.

[49] Q. Tao, W. Wu, C. Zhao, and W. Shen, "An automatic testing approach for compiler based on metamorphic testing technique," in *Proceedings of the 2010 Asia Pacific Software Engineering Conference*, 2010, pp. 270–279.

[50] A. F. Donaldson and A. Lascu, "Metamorphic testing for (graphics) compilers," in *Proceedings of the 1st International Workshop on Metamorphic Testing*, 2016, pp. 44–47.

[51] G. Woo, H. S. Chae, and H. Jang, "An intermediate representation approach to reducing test suites for retargeted compilers," in *Reliable Software Technologies - Ada Europe 2007, Ada-Europe International Conference on Reliable Software Technologies, Geneva, Switzerland, June 25-29, 2007, Proceedings*, 2007, pp. 100–113.

[52] H. S. Chae, G. Woo, T. Y. Kim, J. H. Bae, and W. Y. Kim, "An automated approach to reducing test suites for testing retargeted c compilers for embedded systems," *Journal of Systems & Software*, vol. 84, no. 12, pp. 2053–2064, 2011.

[53] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 192–201.

[54] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: an empirical study," in *Proceedings of the International Conference on Software Maintenance*, 1999, pp. 179–188.

[55] S. Elbaum, A. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2000, pp. 102–112.

[56] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," in *Proceedings of the International Conference on Software Maintenance*, 2001, pp. 92–101.

[57] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing junit test cases," *Transactions on Software Engineering*, vol. 38, no. 6, pp. 1258–1275, 2012.

[58] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test case prioritization: a family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[59] L. Mei, Z. Zhang, W. K. Chan, and T. H. Tse, "Test case prioritization for regression testing of service-oriented business applications," in *Proceedings of the International World Wide Web Conference*, 2009, pp. 901–910.

[60] B. Korel, L. Tahat, and M. Harman, "Test prioritization using system models," in *Proceedings of the International Conference on Software Maintenance*, 2005, pp. 559–568.

[61] Z. Li, M. Harman, and R. Hierons, "Search algorithms for regression test case prioritisation," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[62] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse, "Adaptive random test case prioritization," in *Proceedings of Automated Software Engineering*, 2009, pp. 257–266.

[63] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the International Conference on Software Engineering*, 2001, pp. 329–338.

[64] H. Park, H. Ryu, and J. Baik, "Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing," in *Proceedings of the International Conference on Secure Software Integration and Reliability Improvement*, 2008, pp. 39–46.

[65] S.-S. Hou, L. Zhang, T. Xie, and J. Sun, "Quota-constrained test-case prioritization for regression testing of service-centric systems," in *Proceedings of the International Conference on Software Maintenance*, 2008, pp. 257–266.

[66] J. M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the International Conference on Software Engineering*, 2002, pp. 119–129.

[67] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Time aware test suite prioritization," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2006, pp. 1–11.

[68] L. Zhang, S. Hou, C. Guo, T. Xie, and H. Mei, "Time-aware test-case prioritization using integer linear programming," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2009, pp. 213–224.

[69] M. Qu, M. B. Cohen, and K. M. Woolf, "Combinatorial interaction regression testing: a study of test case generation and prioritization," in *Proceedings of the International Conference on Software Maintenance*, 2007, pp. 255–264.

[70] A. Malishevsky, J. R. Ruthruff, G. Rothermel, and S. Elbaum, "Cost-cognizant test case prioritization," Department of Computer Science and Engineering, University of Nebraska, Tech. Rep., 2006.

[71] H. Do and G. Rothermel, "Using sensitivity analysis to create simplified economic models for regression testing," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2008, pp. 51–62.

[72] ——, "An empirical study of regression testing techniques incorporating context and lifecycle factors and improved cost-benefit models," in *Proceedings of the Symposium on the Foundations of Software Engineering*, Nov. 2006, pp. 141–151.

[73] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009, pp. 201–212.

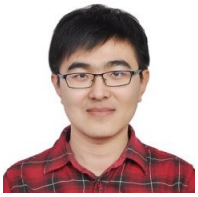
[74] R. Carlson, H. Do, and A. Denton, "A clustering approach to improving test case prioritization: An industrial case study," in *IEEE International Conference on Software Maintenance*, 2011, pp. 382–391.

[75] W. FU, H. YU, G. FAN, and X. JI, "Coverage-based clustering and scheduling approach for test case prioritization," *IEICE TRANSACTIONS on Information and Systems*, vol. 100, no. 6, pp. 1218–1230, 2017.

[76] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 12–22.

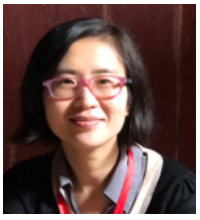
[77] B. Busjaeger and T. Xie, "Learning for test prioritization: an industrial case study," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 975–980.

[78] W. Song, J. Nam, and T. Lin, "Qtep: quality-aware test case prioritization," in *Joint Meeting on Foundations of Software Engineering*, 2017, pp. 523–534.

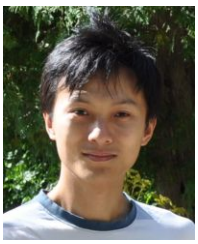


Junjie Chen is a Ph.D. candidate at the School of Electronics Engineering and Computer Science, Peking University. He received his B.S. degree from Beihang University. His research interests are software testing and debugging, mainly focusing on compiler testing, regression testing, automated debugging.

Guancheng Wang is a graduate student for a Master's degree in Department of Computer Science and Technology at Jilin University. With the experience on compiler testing and machine learning research respectively, he is focusing on concatenation work of both.



Dan Hao is an Associate professor at the School of Electronics Engineering and Computer Science, Peking University, P.R. China. She received her Ph.D. in Computer Science from Peking University in 2008, and the B.S. in Computer Science from the Harbin Institute of Technology in 2002. Her current research interests include software testing and debugging. She served on the program committees of many prestigious conferences, such ICSE, ICSME, and so on. She was a general co-chair of SPLC 2018. She is a senior member of ACM.



Yingfei Xiong is an assistant professor under the young talents plan at Peking University. He got his Ph.D. degree from the University of Tokyo in 2009 and worked as a postdoctoral fellow at University of Waterloo between 2009 and 2011. His research interest includes software engineering and programming languages.

Hongyu Zhang is an Associate Professor with The University of Newcastle, Australia. Previously, he was a Lead Researcher at Microsoft Research Asia and an Associate Professor at Tsinghua University, China. He received the PhD degree from National University of Singapore in 2003. His research is in the area of Software Engineering, in particular, software analytics, testing, maintenance, metrics, and reuse. He has published more than 100 research papers in reputable international journals and conferences. He received two ACM Distinguished Paper awards. He has also served as a program committee member for many software engineering conferences. More information about him can be found at: https://sites.google.com/site/hongyujohn/.

Lu Zhang is a professor at the School of Electronics Engineering and Computer Science, Peking University, P.R. China. He received both PhD and BSc in Computer Science from Peking University in 2000 and 1995 respectively. He was a postdoctoral researcher in Oxford Brookes University and University of Liverpool, UK. He served on the program committees of many prestigious conferences, such FSE, ISSTA, and ASE. He was a program co-chair of SCAM2008 and will be a program co-chair of ICSM17. He has been on the editorial boards of Journal of Software Maintenance and Evolution: Research and Practice and Software Testing, Verification and Reliability. His current research interests include software testing and analysis, program comprehension, software maintenance and evolution, software reuse and component-based software development, and service computing.

Bing Xie is a professor at the school of Electronics Engineering and Computer Science, Peking University. He is the director of Software Institute, Peking University. He received his Ph.D. degree from National University of Defense Technology. His research interest includes software engineering, formal method and theory, and distributed system.