Learning to Prioritize Test Programs for Compiler Testing

Junjie Chen[1,2], Yanwei Bai[1,2], Dan Hao[1,2‡], Yingfei Xiong[1,2†‡], Hongyu Zhang[3‡], Bing Xie[1,2]
[1]Key Laboratory of High Confidence Software Technologies (Peking University), MoE
[2]Institute of Software, EECS, Peking University, Beijing, 100871, China
{chenjunjie,byw,haodan,xiongyf,xiebing}@pku.edu.cn
[3]The University of Newcastle, NSW 2308, Australia, hongyu.zhang@newcastle.edu.au

***Abstract*—Compiler testing is a crucial way of guaranteeing the reliability of compilers (and software systems in general). Many techniques have been proposed to facilitate automated compiler testing. These techniques rely on a large number of test programs (which are test inputs of compilers) generated by some test-generation tools (e.g., CSmith). However, these compiler testing techniques have serious efficiency problems as they usually take a long period of time to find compiler bugs. To accelerate compiler testing, it is desirable to prioritize the generated test programs so that the test programs that are more likely to trigger compiler bugs are executed earlier. In this paper, we propose the idea of *learning to test*, which learns the characteristics of bug-revealing test programs from previous test programs that triggered bugs. Based on the idea of *learning to test*, we propose LET, an approach to prioritizing test programs for compiler testing acceleration. LET consists of a learning process and a scheduling process. In the learning process, LET identifies a set of features of test programs, trains a capability model to predict the probability of a new test program for triggering compiler bugs and a time model to predict the execution time of a test program. In the scheduling process, LET prioritizes new test programs according to their bug-revealing probabilities in unit time, which is calculated based on the two trained models. Our extensive experiments show that LET significantly accelerates compiler testing. In particular, LET reduces more than 50% of the testing time in 24.64% of the cases, and reduces between 25% and 50% of the testing time in 36.23% of the cases.**

I. Introduction

Compiler is one of the most fundamental software tools and almost all software systems rely on it. Therefore, it is vitally important to guarantee the reliability of compilers. Compiler testing is an effective and widely-recognized way of ensuring the correctness of compilers [1].

Over the years, many techniques [2], [1], [3], [4], [5] have been proposed to facilitate automated compiler testing. These techniques rely on some test-generation tools (e.g., CSmith) to generate a large number of test programs (which are test inputs of compilers). Compiler bugs can be detected by running the generated test programs. However, compiler testing still suffers from the serious problem of efficiency. For example, Yang et al. [2] spent three years on detecting 325 C compiler bugs, and Le et al. [1] spent eleven months on detecting 147 C compiler bugs. That is, with existing techniques, compiler testing consumes an extremely long period of time to find only a small number of bugs. Therefore, it is very necessary to accelerate compiler testing.

Since only a subset of test programs are able to trigger compiler bugs [3], [2], intuitively, compiler testing can be accelerated by running these test programs earlier. In other words, test prioritization may be adopted to accelerate compiler testing. However, existing test prioritization approaches can hardly be used to accelerate compiler testing due to the following reasons. The dominant prioritization approaches rely on structural coverage information (e.g., statement coverage and branch coverage) [6], [7], [8], which is collected through regression testing [9], [10]. However, most test programs used in compiler testing are generated on the fly by random test generation tools like CSmith, thus the structural coverage information of these test programs is not available. In other words, these prioritization approaches based on structural coverage cannot be applied to accelerate compiler testing (more discussion in Section V-A). Recently, researchers proposed some input-based prioritization approaches [11], [12], which rely on only test inputs (i.e., test programs in compiler testing) and do not require structural coverage information. However, our experimental results (more details in Section IV-A) show that the existing input-based approaches [11], [12] can hardly accelerate compiler testing because of their low efficiency and effectiveness. In summary, the existing test prioritization approaches cannot accelerate compiler testing.

To accelerate compiler testing, in this paper, we present an idea of *learning to test*, which learns the characteristics of bug-revealing test programs to prioritize new test programs. That is, by learning from the existing test programs that trigger bugs, we model the relationship between the characteristics of the test programs and the discovery of compiler bugs. We then use the model to help us prioritize new test programs that are more likely to trigger bugs quickly. Based on this idea, we develop LET (short for **l**earning to **t**est), a *learning-to-test* approach to accelerating compiler testing. Given a set of new test programs, before using a compiler testing technique to test compilers, LET prioritizes these test programs so that programs that have higher chance to trigger bugs in unit time are executed earlier. In particular, in this paper, we target at C compilers because of the following reasons. First, the quality

†Corresponding author.
‡Sorted in the alphabet order of the last names.

of C compilers is very important since many safety-critical software systems are written in C. Second, many different tools and techniques are available for C compiler testing, allowing us to evaluate our approach in different settings.

More specifically, in our work, we study many existing compiler bugs, and identify a set of features on test programs that are related to bug detection. Using the set of features, we train two models from the existing test programs: (1) a capability model that predicts the probability of a new test program to detect a bug, and (2) a time model that predicts the execution time of a new test program. Given a set of randomly generated test programs, LET prioritizes the programs based on the descendant order of their bug-revealing probabilities in unit time, which is calculated by dividing the predicted bug-revealing probability by the corresponding predicted execution time. In this way, our approach accelerates existing compiler testing techniques, leading to more efficient detection of compiler bugs.

We evaluate LET using two compiler testing techniques (DOL [3] and EMI [1]), two subjects (GCC and LLVM), and two application scenarios (cross-compiler and cross-version scenarios). The evaluation results show that, in terms of time spent on detecting each bug, LET substantially accelerates compiler testing in all settings: LET reduces more than 50% of the testing time in 24.64% of the cases, and reduces between 25% and 50% of the testing time in 36.23% of the cases. We also compare LET with two recent input-based test prioritization approaches, i.e., TB−G [12] and ARP [11]. The experimental results show that LET is more effective and stable than TB−G and ARP for accelerating compiler testing.

To sum up, the major contributions of this paper are as follows:

- The idea of "*learning to test*", which learns from existing test programs to accelerate future test execution.
- The development of LET, a *learning-to-test* approach to prioritizing test programs for accelerating C compiler testing.
- An extensive experimental study confirming the effectiveness of our approach.

II. Approach

Figure 1 presents the overview of our approach, which contains an offline learning process (Section II-A) and an online scheduling process (Section II-B).

A. Learning Process

The key insight of our approach is that programs with certain language features or combinations of language features are inherently difficult to compile or optimize, and such programs are more likely to trigger bugs in compilers. If we can correctly identify these features, we should be able to predict the probability of a test program to trigger bugs and thus execute them earlier. We illustrate this with an example. When implementing C compilers, compiling structs is usually complex and error-prone, as the compiler needs to correctly align structs based on the requirement of the underline operating system, and also needs to find an optimal alignment for efficient space usage. As a matter of fact, there are a large number of bug reports in GCC repository related to structs. For example, Figure 2 shows a bug report[1] for GCC (bug ID is 20127), where "volatile" in the struct is not properly treated when performing tree-optimization—the optimization that manipulates GIMPLE trees. From the bug report we can see several features of the code pieces related to the discovery of the bug, for example, the existence of both struct and violatile, and the number of times a volatile variable is written. The former relates to the existence of program elements (*existence features*) and the latter relates to how these elements are used (*usage features*). We identify a large set of existence and usage features and design methods to obtain them from generated programs. We further use a machine learning algorithm to train a prediction model based on the identified features to predict the probability of a test program to trigger a bug. Moreover, in order to get the bug-revealing probability in unit time, we also use a machine learning algorithm to train a regression model to predict the execution time of each test program. We use the same identified features when training the regression model. In this paper, we call the former model as the capability model and the latter model as the time model. In short, our learning process has three components: identifying features, training a capability model and training a time model.

1) Identifying Features: The identified features are divided into two types. The first type of features, existence features, are concerned with whether certain types of elements exist in the target program. Intuitively, some bugs occur only on certain specific programming elements, thus the existence of these programming elements can serve as features to characterize test programs triggering bugs. For example, bugs in loop optimization occur when test programs have loop statements. More concretely, existence features are defined as four sets: $EXIST = STMT \cup EXPR \cup VAR \cup OP$, where

- $STMT$ is the set of all statement types in C language,
- $EXPR$ is the set of all expression types in C language,
- VAR is the set of all variable types in C language, and
- OP is the set of all operation types in C language.

When there exist at least a program element belonging to the associated type exists, the feature is set to one, otherwise it is set to zero.

The second type of features, usage features, are concerned with how the elements in a program are used. Intuitively, certain bugs may only be triggered when the program elements are used in a specific manner. For example, a bug concerning pointers may only be triggered when the pointer has pointed to muliple addresses, i.e., the size of its alias set must be larger than a threshold. In this paper we utilize a characteristic of the random test generation tool CSmith [2], one of the most widely-used random C program generator. When generating a program, CSmith records a set of usage features from the program, such as the size of alias set, the depth of

[1]https://gcc.gnu.org/bugzilla/show_bug.cgi?id=20127.

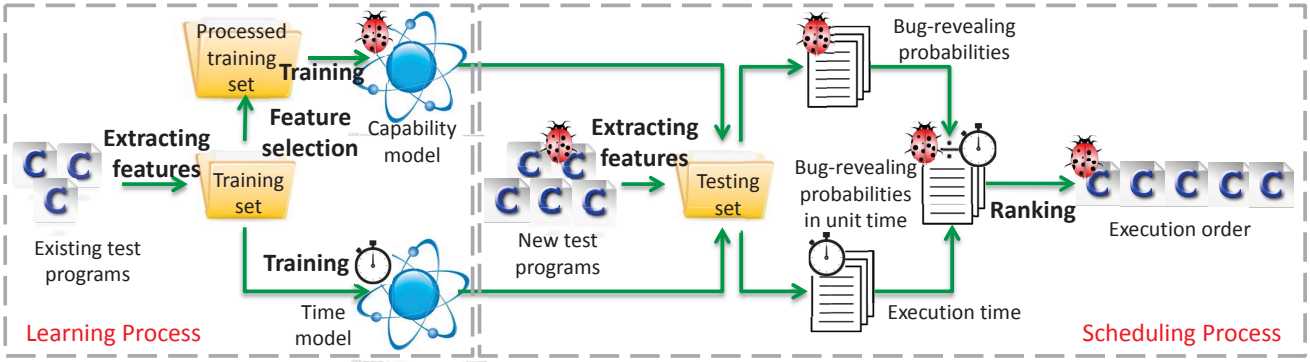

Fig. 1: Overview of LET

```
1  typedef struct{
2     volatile int a;
3     volatile int b;
4  } s;
5  int main (void){
6     s x = {0, 1};
7     s y = {2, 3};
8     x = y;
9     y = x;
10    return x.a + y.a;
11 }
```

Description:
The bug occurs in the "tree-optimization" part of GCC. The code does not treat volatile struct members as volatile, because SRA creates new variables and then goes and makes them renamed.

Fig. 2: An example of GCC bug report

pointer dereference, etc. To save the feature collection time, we directly use the usage features collected by CSmith for our offline training. More concretely, we use the following features:

- Address features, e.g., the number of times the address of a struct or a variable is taken.
- Struct bitfield features, e.g., the times of a struct with bitfields on LHS/RHS, and the number of non-zero, zero, const, violate, full bitfields.
- Pointer dereference features, e.g., the times of a pointer is dereferenced on LHS/RHS, and the depth of pointer dereference.
- Pointer comparison features, e.g., the number of times a pointer is compared with NULL, with the address of another variable, or with another pointer.
- Alias set features, e.g., the size of alias sets.
- Jump features, e.g., the times of forward jumps and backward jumps.
- Used variable features, e.g., the percentage of a fresh-made variable (i.e. the variable defined and used in the same statement) is used and the percentage of an existing variable is used.

Currently, we use only the preceding two types of features due to the tradeoff between acceleration effectiveness and cost. Intuitively, the more features we use in training, the better prioritization results we may get. However, besides the learning process, in the scheduling process LET also needs to extract the values of these features for new test programs. Since the scheduling process is conducted online, it is necessary to control its cost. Therefore, LET uses only these easy-to-extract features.

2) Training a Capability Model: We collect a set of test programs generated by existing test program generation tools, some of which trigger bugs whereas the others do not. Each test program is taken as a training instance whose label is true/false (triggering bugs or not). For each test program, we extract the values of the identified features through program analysis[2]. Based on the set of training instances (including their features and labels), LET first conducts feature selection so as to filter useless features, and then normalizes them in order to adjust values measured on different scales to a common scale, and finally builds a capability model through machine learning. Finally, the capability model outputs the probability of a test program triggering bugs.

- **Feature Selection.** LET conducts feature selection by calculating the information gain ratio of each feature. Information gain ratio is a ratio of information gain to the intrinsic information, which is usually a good measure for identifying the contribution of a feature and is able to reduce the bias towards multi-valued features in existing feature selection metrics [13]. After calculating the information gain ratio of each feature, LET filters the useless features, i.e., the features whose information gain ratios are zero.
- **Normalization.** Since the features are either numeric type or Boolean type (i.e., 0 or 1), LET normalizes each value of these features into the interval $[0, 1]$ using min-max normalization [14]. Supposed the set of new test programs to be scheduled 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:

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

- **Building the Capability Model.** After feature selection and normalization, we adopt a machine learning algorithm,

[2]As our approach is implemented to accelerate compiler testing by using the test programs generated by CSmith [2], we directly extract the values of features (including the two types of features) from each test program during the test-generation process of CSmith.

Sequential Minimal Optimization (abbreviated as SMO) algorithm [15], to build the capability model. The SMO algorithm is a support vector machine algorithm, which speeds up standard support vector machines by breaking a very large quadratic programming optimization problem into a series of smallest possible quadratic programming optimization problems [15].

3) Training a Time Model: We collect a set of test programs and record the execution time of each test program. The execution time of a test program includes the time for compiling the program, the time for running the program and obtaining its result, and the time for any oracle checks necessarily to be performed. We use the previous version of the compiler under test to obtain the execution time. Each test program is taken as a training instance whose label is its execution time. Similar with training the capability model, we also extract the values of the identified features and then normalize the set of training instances. Based on the set of normalized training instances, LET builds a regression model (i.e., the time model) using Gaussian processes since their labels are in a continuous domain. Gaussian process uses lazy learning and a measure of the similarity between points (i.e., the kernel function) to predict the value for an unseen point from the training set [16].

B. Scheduling Process

Based on the learned capability model and time model, LET schedules the execution order of new test programs through the scheduling process. Initially, LET extracts the values of the aforementioned features from each new test program, and uses the two models to predict the bug-revealing probability and execution time for each test program, respectively. Then, LET calculates the bug-revealing probability in unit time for each test program, by dividing the bug-revealing probability (predicted by the learned capability model) with the execution time (predicted by the learned time model). Finally, LET prioritizes new test programs based on the descendent order of their bug-revealing probabilities in unit time.

III. Experimental Study

In the study, we address the following research questions.

- **RQ1**: How effective is LET in accelerating C compiler testing?
- **RQ2**: How does LET perform when being applied to different compiler testing techniques (i.e., DOL and EMI)?
- **RQ3**: How does LET perform in different application scenarios (i.e., cross-compiler and cross-version scenarios)?
- **RQ4**: Can the major components of LET (i.e., feature selection and time model) contribute to the overall effectiveness?

More specifically, RQ1 investigates overall acceleration effectiveness of LET by comparing it with two existing prioritization approaches, RQ2 investigates the effectiveness of LET on accelerating different compiler testing techniques, RQ3 investigates two application scenarios which differ in the subjects used in the learning process and the scheduling process (more details referred to Section III-B). Furthermore, for LET, its key part is to train a capability model using the identified features from existing test programs. To improve the effectiveness of its key part, there are two important complementary parts: feature selection and time model. Therefore, RQ4 studies whether feature selection and time model make contributions to the effectiveness of LET, respectively.

TABLE I: Subject statistics

Subject	LOC	Usage
GCC-4.3.0	3,343,377	Learning
GCC-4.4.3	4,727,209	Scheduling
LLVM-2.6	684,114	Learning & Scheduling
LLVM-2.7	795,152	Scheduling
Open64-5.0	6,078,400	Learning

A. Subjects and Test Programs

In this experimental study, we use three mainstream open-source C compilers, namely GCC, LLVM, and Open64 for the x86_64-Linux platform.

The statistics of these compilers are presented in Table I, where the last column presents whether the corresponding compiler is used in training the capability model (marked as *Learning*) or is used in testing the approach (marked as *Scheduling*). Note that we always use the previous version to train the time model. The subjects we use are not the newest versions, because it is easier for us to collect enough bugs on the old versions to perform statistical analysis. We only use Open64 in the learning process because it has no available bug reports, making it impossible to measure the number of bugs detected. More discussion can be found at III-E.

The test programs we use in both learning and scheduling processes are C programs randomly generated by CSmith [2], which is commonly used in the literature of C compiler testing [1], [3]. The programs generated by CSmith are always valid and do not require external inputs, and the output of any program is the checksum of the non-pointer global variables of the program at the end of program execution. To avoid imbalance data problem in building a capability model, for each subject used in the learning process, we randomly collect the same number of test programs triggering bugs and test programs not triggering bugs[3].

B. Application Scenarios

We consider two application scenarios of LET in this study.

Cross-compiler scenario: LET learns a capability model from one compiler and applies the capability model to prioritize test programs for another compiler. To evaluate LET in this scenario, we use Open64-5.0 in the learning process and use GCC-4.4.3 as well as LLVM-2.6 respectively in the scheduling process, and use GCC-4.3.0 in the learning process and use LLVM-2.6 in the scheduling process.

Cross-version scenario: LET learns a capability model from one version of a compiler and applies the capability model to prioritize test programs for its later versions. To

[3]For Open64-5.0 and GCC-4.3.0, we use 1000 failed test programs and 1000 passed test programs. For LLVM-2.6, we use 800 failed test programs and 800 passed test programs because CSmith do not generate enough failed test programs.

evaluate LET in this scenario, we use GCC-4.3.0 in the learning process and GCC-4.4.3 in the scheduling process, and use LLVM-2.6 in the learning process and LLVM-2.7 in the scheduling process. In particular, in this scenario, all the bugs used in the learning process have already been resolved and closed before the versions used in the scheduling process.

C. Baseline Approach

Random order approach (RO), which randomly selects an execution order of new test programs, is taken as the baseline in our study. RO demonstrates the effectiveness of compiler testing without any accelerating approaches.

D. Independent Variables

We consider three independent variables in the study.

1) Compiler Testing Techniques: In this study, we consider two compiler testing techniques to accelerate, i.e., Different Optimization Level (DOL) [3] and Equivalence Modulo Inputs (EMI) [1].

DOL is an effective compiler testing technique, which determines whether a compiler contains bugs by comparing the results produced by the same test program with different optimization levels (i.e., -O0, -O1, -Os, -O2 and -O3) [3]. Given an execution order decided by a prioritization 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 is first proposed by Le et al. [1], which generates some equivalent variants for any given test program and determines whether a compiler contains bugs by comparing the results produced by the original test program and its variants[4]. Given an execution order decided by a prioritization approach, we generate eight variants for each original test program by randomly deleting its unexecuted statements as EMI did, 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.

2) Compared Prioritization Approaches: We implement two latest input-based test prioritization approaches for comparison.

Token-vector based prioritization (TB−G) [12], which is the first test case prioritization approach for compilers. TB−G regards each test program as text and transforms each test program into a text-vector by extracting corresponding tokens from text, and then prioritizes test programs based on the distance between the text-vector and the origin vector (0, 0, ..., 0). As an existing study [12] reveals, TB−G is the most cost-effective strategy among a set of studied techniques.

Adaptive random prioritization (ARP), which selects the test input that has the maximal distance to the set of already selected test inputs [11]. Although ARP is not proposed to accelerate compiler testing, in our study, we adopt it for compiler testing by treating a test program as a test input and calculating the distance between test programs using their edit distance.

3) Variants of LET: In our study we explore the impact of some parts (i.e., feature selection and time model) of LET on compiler testing acceleration, thus we implement the following variants of LET.

LET−A, which removes the feature selection process (described in Section II-A2) from LET. That is, LET−A uses all identified features to train the capability model, and then trains a time model, and finally prioritizes new test programs based on their bug-revealing probabilities in unit time.

LET−B, which removes the time model (as described in Section II-A3) from LET. That is, LET−B conducts feature selection to filter useless features, and then uses the processed training set to build a capability model, and finally prioritizes new test programs based on only their bug-revealing probabilities.

To implement LET, LET−A and LET−B, we use Weka 3.6.12 [19], which is a popular environment for data mining. We use the SMO algorithm implemented by Weka to build the capability model, choosing Puk kernel with $omega = 1.0$ and $sigma = 0.7$. We use the Gaussian process implemented also by Weka to build the time model, choosing Puk kernel with $omega = 3.3$ and $sigma = 0.5$. 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 Weka.

E. Dependent Variables

An important issue is how we measure the number of bugs detected by a test suite. If two test programs both fail, there may be two bugs, or the two test programs may trigger the same bug. To solve this problem, we use the Correcting Commits technique used in previous work [3]. That is, given a failed test program, we determine in which future commit the bug is corrected, i.e., the test program passes since that commit. If two bugs are corrected by the same commit, we assume the two bugs are the same bug.

The dependent variable considered in our study is the time spent on detecting k bugs, where $1 \leq k \leq n$ and n is the total number of bugs detected when executing all test programs in our study. This dependent variable is used to measure the effectiveness of LET. Note that we do not use the average percentage of detected faults (abbreviated as APFD) [20], [21], [12], because developers usually care more about the time spent in compiler testing rather than the number of test programs used in compiler testing.

For ease of presentation, we use Formula 1 to calculate the corresponding speedup on detecting k bugs, where $TRO(k)$ represents the time spent on detecting k bugs through RO (i.e., without any accelerating approaches), and $TACC(k)$ represents the time spent on detecting k bugs using a prioritization approach (i.e., LET, LET−A, LET−B, TB−G, or ARP).

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

[4]In fact, EMI has three instantiations, namely Orion [1], Athena [17], and Hermes [18]. In our paper, EMI refers to Orion.

Since all the accelerating approaches (e.g., LET, TB−G and ARP) studied in this paper consume extra time on scheduling test programs[5], the time spent on detecting k bugs includes the scheduling time.

F. Experimental Process

For compilers under test, we generate 100,000 C programs using CSmith, which serve as the new test programs to be scheduled.

First, we apply RO to the test programs and feed the scheduled 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 triggered 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 LET to each compiler under test in two application scenarios by using two compiler testing techniques. During this process, we also calculate the time spent on detecting each bug. Following the same procedure we apply LET−A and LET−B.

Finally, we apply TB−G and ARP to the new test programs of compilers under test and feed the prioritized test programs to the two compiler testing techniques respectively, calculating the time spent on detecting each bug.

The experimental study is conducted on a workstation with eight-core Intel Xeon E5620 CPU with 24G memory, and Ubuntu 14.04 operating system.

G. Threats to Validity

The threats to internal validity mainly lie in the implementations of our approach, the compared approaches and the compiler testing technique EMI. To avoid implementation errors, at least two authors of this paper review the source code. Furthermore, in implementing EMI, we use the same tools (i.e., LibTooling Library of Clang[6] and Gcov[7]) as Le et al. [1] did in their implementation.

The threats to external validity mainly lie in compilers and test programs. To reduce the threat resulting from compilers, we use all the C compilers that have been used in the literature on compiler testing [2], [1], [3]. In the future, we will use more compilers and versions as subjects. To reduce the threat resulting from test programs, we use C test programs randomly generated by CSmith as the prior work did [1], [2], [3]. However, these test programs are not necessarily representative of C programs generated by other tools. In the future, we will further use more other test generation tools.

The threats to construct validity lie in how the results are measured. In measuring acceleration effectiveness, we use Correcting Commits to automatically identify duplicated bugs [3]. As Correcting Commits relies on developers' edition, different bugs may be regarded as the same one. However, it may not be a big threat because developers do not tend to fix many of bugs in one commit to guarantee software quality [3].

H. Verifiability

The replication package of the experiments is available at our project website[8]. The package includes the tool and the data for reproducing the experiments. It also includes the detailed experimental results. The tool is open source, allowing one to verify the details of the experiments. The detailed results allow one to verify the result analysis without rerunning the experiments. The tool also enables further studies on different subjects and different experimental settings.

IV. Experimental Results and Analysis

A. RQ1: Acceleration Effectiveness

Table II presents the acceleration results of LET and TB−G in terms of the time spent on detecting bugs. In this table, Column "Scenarios" presents the application scenarios, and the subject before an arrow is used in the learning process, whereas the subject behind the arrow is used in the scheduling process, Column "Bug" presents the number of detected bugs, Column "RO" presents the average time spent on detecting the corresponding number of bugs, Columns "ΔLET" and "ΔTB−G" present the difference between the time spent on detecting the corresponding number of bugs using LET/TB−G and using RO. *If the difference is less than zero, the corresponding approach accelerates compiler testing* because the used accelerating approach (LET or TB−G) spends less time than RO on detecting the corresponding number of bugs.

• **Overall Effectiveness** Table II shows that the values in Column "ΔLET" are mostly smaller than zero. Moreover, the absolute values that are smaller than zero are far larger than the absolute values that are larger than zero in most cases, e.g., in the scenario Open64-5.0 → GCC-4.4.3 using DOL, the absolute values that are smaller than zero includes 6.38 and 4.92 but the absolute values that are larger than zero are only 0.08 and 0.17. Furthermore, as the testing proceeds, the time spent on testing will become larger, and the absolute saving of LET will become larger even if the relative speedup stays the same. Overall, LET does accelerate compiler testing.

We also analyze the distribution of speedups achieved by LET[9]. The results are shown in Figure 3. In this figure, the x-axis represents the scope of speedups and the y-axis represents the frequency in which speedups occur corresponding scopes. From the left bar for each scope in this figure, we can see that in more than 88.41% cases LET accelerates compiler testing, and the speedups of LET are mostly in the range from 25% to 50% (medium acceleration) and secondly in the range more than 50% (high acceleration). That is, LET does accelerate compiler testing to a great extent.

[5]In our approach, the extra time refers to only the time spent on the scheduling process described in Section II-B because the learning process is conducted offline.

[6]http://clang.llvm.org/docs/LibTooling.html.

[7]http://ltp.sourceforge.net/coverage/gcov.php.

[8]https://github.com/JunjieChen/let.

[9]For each k and each setting, we calculate the speedup of LET using Formula 1. We then analyze these speedups together.

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

Scenarios	Bug	DOL			EMI			Scenarios	Bug	DOL			EMI		
		RO	ΔLET	ΔTB−G	RO	ΔLET	ΔTB−G			RO	ΔLET	ΔTB−G	RO	ΔLET	ΔTB−G
Open64-5.0 → GCC-4.4.3	1	0.02	0.00	0.40	0.18	-0.14	2.43	GCC-4.3.0 → GCC-4.4.3	1	0.02	0.02	0.39	0.29	0.64	2.32
	2	0.15	-0.07	0.28	0.70	-0.65	3.18		2	0.32	-0.07	0.11	0.78	1.78	3.09
	3	0.60	-0.03	1.30	2.77	-2.69	5.89		3	0.89	-0.56	1.01	1.43	1.45	7.24
	4	1.02	-0.39	1.80	4.18	-3.88	4.48		4	1.90	-1.12	0.91	2.98	0.58	5.68
	5	1.72	-0.87	1.20	6.76	-5.37	1.90		5	2.36	-1.17	0.56	5.13	0.67	3.53
	6	3.00	-1.88	0.00	9.22	-7.57	-0.56		6	2.93	-1.51	0.08	9.35	-3.55	-0.69
	7	4.68	-1.90	-0.16	10.49	-8.84	-1.04		7	3.96	-2.48	0.55	12.21	-1.24	-2.76
	8	5.25	-1.58	-0.53	14.52	-12.86	-4.83		8	4.82	-1.84	-0.10	14.66	-3.05	-4.98
	9	7.72	-3.33	-2.08	19.74	-18.09	-3.23		9	6.31	-2.15	-0.66	19.35	-0.88	-2.83
	10	8.66	-4.19	-1.65	24.55	-22.40	-4.77		10	7.58	-3.07	-0.57	23.25	-3.90	-3.47
	11	10.46	-3.80	-0.23	28.64	-25.31	2.08		11	9.67	-5.15	0.56	27.26	-7.48	3.46
	12	12.22	-3.43	2.08	32.97	-25.39	0.19		12	12.11	-6.73	2.19	29.69	-9.91	3.47
	13	14.23	-4.92	0.71	40.55	-31.96	-2.34		13	14.73	-6.85	0.21	36.02	-13.73	2.19
	14	16.11	-6.38	0.14	49.30	-40.07	-5.35		14	17.47	-6.47	-1.22	40.54	-18.24	3.42
	15	18.06	-4.19	3.22	55.36	-45.33	-10.45		15	18.92	-4.34	2.37	48.86	-16.01	-3.95
	16	19.40	-0.57	2.06	61.76	-46.45	-14.32		16	21.08	-5.62	0.38	53.90	-11.08	-6.45
	17	22.18	0.17	3.75	72.24	-38.93	-23.53		17	24.37	-4.31	1.55	58.08	-11.64	-9.37
	18	23.59	0.08	2.67	74.82	-23.50	-23.74		18	26.39	-4.37	-0.13	61.69	-15.26	-10.61
	19	28.01	-1.09	3.42	79.83	-17.15	-28.75		19	29.63	-4.99	1.80	76.52	-23.99	-25.44
	20	32.16	-3.09	1.16	90.34	-20.48	-33.80		20	32.08	-6.77	1.24	78.96	-21.61	-22.42
	21	—	—	—	100.05	-30.15	-32.63		21	—	—	—	88.59	-28.98	-21.17
	22	—	—	—	106.20	-15.76	-37.37		22	—	—	—	107.05	-34.01	-38.22
	23	—	—	—	115.80	-25.36	-43.80		23	—	—	—	115.04	-35.63	-43.04
Open64-5.0 → LLVM-2.6	1	0.35	-0.15	1.08	3.32	-1.96	-0.23	GCC-4.3.0 → LLVM-2.6	1	0.35	-0.06	1.08	3.32	-0.29	-0.23
	2	1.06	-0.09	0.69	8.56	-4.79	-4.02		2	1.06	-0.73	0.69	8.56	-3.76	-4.02
	3	2.60	-0.14	-0.29	16.08	-7.37	-5.21		3	2.60	-1.54	-0.29	16.08	-11.28	-5.21
	4	4.33	-1.66	-1.05	23.25	-11.27	-4.36		4	4.33	-2.87	-1.05	23.25	-3.97	-4.36
	5	5.23	-2.33	-0.91	33.23	-9.46	-13.16		5	5.23	-1.63	-0.91	33.23	-13.95	-13.16
	6	6.83	-1.92	-2.43	47.86	6.67	3.88		6	6.83	-0.01	-2.43	47.86	-23.19	3.88
	7	8.73	-3.43	1.66	56.30	6.17	11.63		7	8.73	1.12	1.66	56.30	-29.80	11.63
	8	9.97	-3.26	0.82	66.83	14.21	5.84		8	9.97	3.11	0.82	66.83	-27.50	5.84
	9	13.37	-6.65	-1.14	86.07	-5.03	-9.40		9	13.37	-0.06	-1.14	86.07	-38.23	-9.40
	10	16.33	-7.62	1.38	94.70	-11.19	-18.03		10	16.33	-0.53	1.38	94.70	-43.85	-18.03
	11	20.25	-1.60	-2.05	—	—	—		11	20.25	-0.84	-2.05	—	—	—
	12	22.87	2.60	-1.71	—	—	—		12	22.87	-2.85	-1.71	—	—	—
	13	25.45	4.46	-1.72	—	—	—		13	25.45	-5.07	-1.72	—	—	—
	14	30.76	0.16	-3.19	—	—	—		14	30.76	-9.60	-3.19	—	—	—
LLVM-2.6 → LLVM-2.7	1	1.35	-0.69	0.21	3.19	-0.38	-0.47	—	–	—	—	—	—	—	—
	2	—	—	—	22.95	-20.14	-13.30		–	—	—	—	—	—	—
	3	—	—	—	61.84	-23.14	-3.70		–	—	—	—	—	—	—

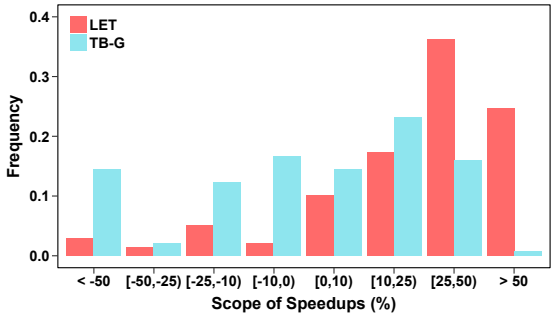

Fig. 3: Speedup distribution of LET

• **Comparison with TB−G** Table II shows that the values in Column "ΔLET" are smaller than those in Column "ΔTB−G" in most cases, thus LET spends less time on detecting the same number of bugs than TB−G in most cases. Besides, the former are mostly smaller than zero, whereas the latter are often larger than zero. Therefore, LET performs more stable than TB−G in accelerating compiler testing.

Similarly, we also analyze the distribution of speedups (calculated by Formula 1) achieved by TB−G, whose results are also shown in Figure 3. From the right bar for each scope in this figure, in about 54.35% cases TB−G accelerates compiler testing and in 45.65% cases it decelerates compiler testing. Moreover, the speedups achieved by TB−G are mostly in the range from 10% to 25% (low acceleration) and then the range from -10% to 0%. In particular, in only 0.72% cases the speedups achieved by TB−G are in the range more than 50% (high acceleration). Therefore, LET performs much better and more stable than TB−G in accelerating compiler testing.

To learn whether LET outperforms TB−G significantly, we perform a paired sample Wilcoxon signed-rank test (at the significance level 0.05), whose results are shown in Table III. Since the number of bugs detected in LLVM-2.7 using either DOL or EMI is quite small during the corresponding testing periods in our study[10], we cannot perform this statistical test on those settings. In this table, Rows "p-value" represent the p-value of the respective scenarios, which reflect the significance in statistics. The p-values with (✚) denote that LET outperforms TB−G significantly in the corresponding setting. Besides, Row "Mean(%)" represents the mean speedups between LET and TB−G, which is calculated by adapting Formula 1 where $TRO(k)$ refers to the time spent on detecting k bugs through TB−G and $TACC(k)$ refers to the time spent on detecting k bugs through LET. From this table, LET outperforms TB−G in all settings and mean improvements range from 12.16% to 79.97%, and LET outperforms TB−G significantly at five settings. Therefore, LET outperforms TB−G significantly in majority cases.

In conclusion, LET does perform much better and more stable than TB−G in accelerating compiler testing.

[10]DOL detects only one bug and EMI detects only three bugs.

TABLE III: Statistical analysis between LET and TB−G

Scenarios		Open64-5.0 → GCC-4.4.3	Open64-5.0 → LLVM-2.6	GCC-4.3.0 → GCC-4.4.3	GCC-4.3.0 → LLVM-2.6
DOL	Mean(%)	50.91	29.48	51.25	29.59
	p-value	0.000(✚)	0.358	0.000(✚)	0.217
EMI	Mean(%)	79.91	12.16	24.56	32.84
	p-value	0.015(✚)	0.625	0.012(✚)	0.020(✚)

TABLE IV: Comparison between Different Compiler Testing Techniques and Application Scenarios

Summary	Techniques		Scenarios	
	DOL	EMI	Cross-compiler	Cross-version
Mean(%)	30.91	50.81	47.85	27.69
p-value	0.000(✚)	0.000(✚)	0.000(✚)	0.000(✚)

• **Comparison with ARP** In our experiment, ARP does not accelerate compiler testing at all, which is consistent with the existing study [12]. In particular, with ARP, DOL detects only one bug of GCC-4.4.3 in $30*10^4$ seconds (i.e., about the total execution time of all test programs using DOL) and detects no bugs of LLVM-2.6 or LLVM-2.7 by DOL in the same period. ARP performs even worse on EMI, because with ARP no bug is detected in GCC-4.4.3, LLVM-2.6 and LLVM-2.7 in $100*10^4$ seconds (i.e., about the total execution time of all test programs using EMI). The reason ARP performs worse on accelerating compiler testing is that it spends too much time on calculating the edit distance between test programs whenever selecting a new test program. Therefore, although ARP does not rely on structural coverage information and is reported to achieve acceptable effectiveness in general software [11], it cannot be applied to accelerate C compiler testing.

B. RQ2: Impact of Compiler Testing Techniques

To answer RQ2, we statistically analyze all the experimental results of LET for each compiler testing technique ignoring the impact of subjects and application scenarios. The analysis results are given by the first three columns of Table IV, where row "Mean(%)" presents the mean speedups of LET and the row "p-value" presents the p-values of the paired sample Wilcoxon signed-rank test between LET and RO.

From this table, LET accelerates compiler testing significantly regardless of using either DOL or EMI. Moreover, the acceleration effectiveness of LET using any of DOL and EMI is quite obvious, namely, their mean speedups are 30.91% and 50.81% respectively. Therefore, LET achieves great effectiveness for accelerating different compiler testing techniques.

C. RQ3: Impact of Application Scenarios

To answer RQ3, we also statistically analyze the results of LET for each application scenario ignoring the impact of subjects and compiler testing techniques. The analysis results are given by the latter two columns of Table IV. From this table, in either the cross-compiler scenario or the cross-version scenario, LET accelerates compiler testing significantly. Moreover, the acceleration effectiveness of LET in any of the two application scenarios is quite obvious, namely, their mean speedups are 47.85% and 27.69% respectively. This is an evidence to demonstrate the soundness of LET. That is, LET accelerates compiler testing no matter which compiler or version is used to train the models.

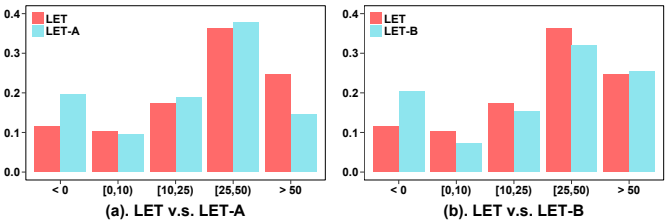

Fig. 4: Comparison between LET and its variants with respect to the distribution of speedups

D. RQ4: Contributions of Major Components of LET

Figures 4 presents the comparison between LET and its variants (i.e., LET−A and LET−B). In these figures, the x-axis represents the scope of speedups and the y-axis represents the frequency in which speedups occur corresponding scopes. From Figure 4(a), the number of deceleration cases by LET−A is larger than that by LET. That is, LET is more stably effective than LET−A. Moreover, the speedups of LET−A are mostly in the range from 25% to 50% and the range less than 0%, but the speedups of LET are mostly in the range from 25% to 50% and the range more than 50%. That is, LET performs better than LET−A. Therefore, feature selection does improve the acceleration effectiveness of LET by filtering useless features.

Furthermore, we identified the features that contribute more to bug detection through the analysis of information gain ratio. These features include the number of times a non-volatile is written, the maximum expression depth, the existence of struct type, the number of struct whose depth is one, etc. Some features are expected, such as the existence of struct type, which is consistent with the example presented in Figure 2. However, some features are a bit surprising to us, such as the number of struct whose depth is one. Intuitively, test programs with larger depth of struct tend to be more complex, thus they are more likely to detect compiler bugs. But our evaluation finds that, when the depth of struct is one, it is more likely to detect compiler bugs. We will investigate the relationship between features and bug-triggering ability in our future work.

Similarly, from Figure 4(b), the number of deceleration cases by LET−B is larger than that by LET. That is, LET is more stably effective than LET−B. Furthermore, although the speedups of both LET and LET−B are mostly in the range from 25% to 50% and the range more than 50%, the total frequency in those scopes of LET, i.e., 60.87%, is larger than that of LET−B, i.e., 57.25%. Moreover, the frequencies in the range from 10% to 25% and the range from 0% to 10% of LET is also larger than those of LET−B respectively. That is, LET performs better than LET−B. Therefore, training a time model also does improve the acceleration effectiveness of LET by prioritizing test programs based on their bug-revealing probabilities in unit time.

Overall, feature selection and time model actually improve the prioritization effectiveness, and make contributions to the acceleration effectiveness of LET.

V. Discussion

A. Coverage-based Prioritization Doesn't Work

In this paper, we did not investigate existing coverage-based prioritization approaches because they can hardly be applied to accelerate compiler testing. Typically, coverage-based prioritization schedules the execution order of tests for the project under test by utilizing the coverage information of its previous version. That is, coverage-based prioritization approaches are proposed based on a hypothesis that many tests designed for the previous version can be reused to detect bugs for the current version. However, this hypothesis may not hold in compiler testing. To verify this hypothesis, we conduct a preliminary study on GCC using DOL. In particular, we randomly generate 20,000 test programs by using CSmith and run these programs on GCC-4.3.0, recording the bugs detected by DOL. Then we search the subsequent versions of GCC-4.3.0 to find its closest version (i.e., GCC-4.4.0) that fixes all the detected bugs. To verify whether the 20,000 test programs are still useful in detecting bugs for new versions, we run these test programs on GCC-4.4.3, which is a later version of GCC-4.4.0, and find no bug at all. On the contrary, we generate another 20,000 new test programs using CSmith and find that 295 new programs reveal 11 bugs in GCC-4.4.3. Comparing the bugs detected by 20,000 old programs and 20,000 new programs in GCC-4.4.3, we can tell that, in compiler testing, new test programs may outperform old test programs. Similarly, Le et al. [1] also demonstrated new test programs perform better than the regression test suite using EMI. Moreover, there are mature program generation tools like CSmith, which can generate a large number of test programs efficiently. Therefore, practical compiler testing usually uses new test programs rather than reuse old test programs. Besides, existing compiler testing techniques [2], [1], [3] also use new test programs to test compilers.

For new test programs, it is difficult to acquire their coverage information from a previous compiler version. Moreover, we can hardly collect such coverage information without running the new test programs. Therefore, coverage-based prioritization can hardly be applied to accelerate compiler testing and we do not compare with these approaches in this study.

B. Training Efficiency of LET

LET needs to train two models: a capability model and a time model. To train a capability model, LET uses a fixed previous version of a compiler or another compiler so as to reduce the cost of retraining a capability model. That is, in practical usage, when a capability model has been trained, it can be used to test a series of versions or compilers. From the results of our study, when a capability model is trained from GCC-4.3.0, LET accelerates the testing of GCC-4.4.3 and LLVM-2.6 using the capability model; when a capability model is trained from Open64-5.0, LET also accelerates the testing of GCC-4.4.3 and LLVM-2.6 using the capability model. That is, the trained capability model is indeed robust. In particular, even if we need to retrain a capability model, the cost is very small compared with the testing time we reduce. In our experiments, training a capability model takes less than two minutes, while the trained model on average reduced about 44 hours in test execution time.

To train a time model, LET uses the previous version of the version under test as the training version in order to make the prediction results more accurate. We assume frequent retraining on time model because the cost of retraining is even smaller than that of the capability model: to train a capability model, we need at least a set of programs triggering bugs, but to train a time model, the training programs and their labels can be collected when testing the previous version. Furthermore, the retraining cost of a time model is also less than two minutes. On the other hand, even if a time model is not available, LET-B still significantly accelerates compiler testing.

C. Potential Applications

We believe that the general concept of *learning to test* proposed in this paper has many potential application areas.

First, besides the compiler testing techniques used in the evaluation, LET can be applied to accelerate other compiler testing techniques, e.g., Randomized Differential Testing (RDT) [22] by feeding the prioritized test programs to these techniques. Second, besides C compilers, LET can be applied to compilers of other languages (e.g., Java). In fact, as long as testers identify relevant features, our *learning-to-test* approach can be applied to compilers of other languages directly. Finally, our *learning-to-test* approach can be also applied to other types of complex software besides compilers, e.g., browsers, operating systems and image processing software. Like C compilers, the inputs of complex software tend to be complex too (e.g., images for image processing software), whose complexity provides an opportunity for us to identify different characteristics that are related to detected bugs from them, thus our *learning-to-test* approach can be applied to test other types of complex software.

VI. Related Work

A. Compiler Testing

Compiler testing is difficult because the test inputs of compilers are programs, and the generated programs must strictly meet complex specifications (e.g., C99 specification for C programs). Moreover, compiler testing suffers from the test oracle problem as it is hard to tell the expected outputs of a compiler given some test programs [3], [1], [23], [24], [25]. In the literature, there are two main aspects on compiler testing, namely generating test programs and addressing the test oracle problem.

For test program generation, random test program generation is a main generation technique [26], [27], [25], [28], [29], [30], [31], [32]. For example, Yang et al. [2], [33] proposed and implemented a tool, called CSmith, to randomly generate C programs without undefined behaviors for testing C compilers, and Regehr et al. [34] proposed test case reduction

for C compiler bugs. Zhao et al. [35] developed an integrated tool (JTT), which automatically generates programs to test UniPhier, an embedded C++ compiler. Lidbury et al. [25] developed CLsmith based on CSmith to generate programs to test OpenCL compilers, and Pflanzer et al. [36] proposed test case reduction for OpenCL.

To address the test oracle problem of complex software systems (including compilers), McKeeman et al. [22] 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, which has been widely used to detect various compiler defects [5], [37], [4]. Le et al. [1], [17], [18] proposed to generate some equivalent variants for each original C program, which determines whether a compiler has bugs by comparing the results produced by the original program and its variants. This technique is called Equivalence Modulo Inputs, which has three instantiations: Orion [1], Athena [17], and Hermes [18]. Tao et al. [38] proposed to test compilers by constructing metamorphic relations, e.g., the equivalent relation. Boussaa et al. [39] proposed NOTICE, a component-based framework for non-functional testing of compilers according to user requirements. Furthermore, Chen et al. [3] conducted an empirical study to compare mainstream compiler testing techniques (i.e., RDT, DOL and EMI), and Sun et al. [40] conducted a study to analyze the characteristics of the bugs in GCC and LLVM.

Different from these approaches, our work addresses another important problem in compiler testing, i.e., the test efficiency problem. It is time-consuming to detect compiler bugs, which is a common problem of the existing compiler testing techniques. As the efficiency problem is quite serious to compiler testing, our work targets at accelerating compiler testing.

B. Test Prioritization

In the literature, there is a considerable amount of research on test prioritization [6], [21], [7], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [9], which can be mainly classified into four groups. The first group focuses on the criterion used in test prioritization, including structural coverage criterion [6], [7], [8], [51], the probability of exposing faults [52], dataflow coverage [53] and system model coverage [54]. The second group focuses on the algorithms used in test prioritization, including greedy algorithms (i.e., total and additional strategies [6]) and many meta heuristics algorithms [21], [55]. The third group focuses on the evaluation of existing test prioritization techniques [6], [56], [57], [58], [59], [60], [61], including measurement on the effectiveness of test prioritization techniques and the influence of some factors in test prioritization. The fourth group focuses on constraints that affect test prioritization, e.g., time constraints [57], and the work in this group investigates the influence of the constraints and prioritization techniques specific to some constraints [41], [62], [63], [59], [64]. In this group, time-aware prioritization approaches [59], [64], [65] also utilize execution time of test cases. They use the execution time collected from a previous version in regression testing, while our work predicts the execution time of new test programs for the current version by training a time model.

Similar to existing work on test prioritization, our work targets at prioritizing test cases that are more likely to reveal bugs so as to accelerate compiler testing. However, as discussed in Section V-A, most test prioritization approaches are based on coverage information and cannot be applied to accelerate compiler testing. Recently, Jiang et al. [11] proposed an adaptive random prioritization technique, which prioritizes the execution order of test cases based on only test inputs rather than coverage information. This approach is motivated by adaptive random testing [66], [67], which is a test generation technique to spread the test inputs evenly in the input domain. Chen et al. [12] proposed a text-vector based prioritization approach, which transforms each test case into a text-vector by extracting corresponding tokens. The two approaches may be adopted to accelerate compiler testing, but in our study their acceleration effectiveness is not satisfactory and in many cases they even decelerate compiler testing. In this paper, we propose a *learning-to-test* approach to accelerating compiler testing, which achieves good and stable acceleration performance.

C. Software Defect Prediction

There is a large amount of research on software defect prediction [68], [69], [70], [71], [72], [73], [74], [75], [76], [77], [78], [79], which predicts defect-prone programs (and code modules) by training classifiers based on the features extracted from historical defective code. Different from defect prediction, our work targets at predicting the bug-revealing probabilities of test programs, not the defect-proneness of test programs themselves.

VII. Conclusion

In this paper, we propose the idea of *learning to test*, which utilizes the characteristics of existing test cases that trigger bugs. Based on this idea, we develop LET, a *learning-to-test* approach to accelerating C compiler testing. This approach has two processes: learning and scheduling. In the learning process, LET identifies a set of features of bug-revealing test programs and trains a capability model and a time model. In the scheduling process, LET ranks new test programs based on the two models. We evaluate our approach using two compiler testing techniques (i.e., DOL and EMI), two subjects (i.e., GCC and LLVM) and two application scenarios (i.e., cross-compiler scenario and cross-version scenario). Our experimental results demonstrate that LET accelerates C compiler testing significantly in all settings.

VIII. Acknowledgment

This work is supported by the National Key R&D Program of China under Grant No.2016YFB1000801, and the National Natural Science Foundation of China under Grant No. 61672047, 61522201, 61421091, 61272089.

REFERENCES

[1] 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.

[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] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] D. Hao, L. Zhang, and H. Mei, "Test-case prioritization: achievements and challenges," *Frontiers of Computer Science*, vol. 10(5), pp. 769–777, 2016.

[10] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, and B. Xie, "How do assertions impact coverage-based test-suite reduction?" in *Proceedings of the 10th International Conference on Software Testing, Verification and Validation*, 2017, to appear.

[11] B. Jiang and W. K. Chan, "Input-based adaptive randomized test case prioritization: A local beam search approach," *Journal of Systems and Software*, vol. 105, pp. 91–106, 2015.

[12] 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.

[13] 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.

[14] Y. K. Jain and S. K. Bhandare, "Min max normalization based data perturbation method for privacy protection," *International Journal of Computer & Communication Technology*, vol. 2, no. 8, pp. 45–50, 2011.

[15] J. C. Platt, "Fast training of support vector machines using sequential minimal optimization," *Advances in kernel methods*, pp. 185–208, 1999.

[16] C. E. Rasmussen, "Gaussian processes in machine learning," in *Advanced lectures on machine learning*, 2004, pp. 63–71.

[17] 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.

[18] 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.

[19] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[20] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, and T. Xie, "To be optimal or not in test-case prioritization," *IEEE Transactions on Software Engineering*, vol. 42, no. 5, pp. 490–505, 2015.

[21] 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.

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

[23] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, B. Xie, and H. Mei, "Supporting oracle construction via static analysis," in *Proceedings of the 31st International Conference on Automated Software Engineering*, 2016, pp. 178–189.

[24] 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.

[25] 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.

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

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

[28] F. Sheridan, "Practical testing of a c99 compiler using output comparison," *Software: Practice and Experience*, vol. 37, no. 14, pp. 1475–1488, 2007.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," in *PLDI*, 2016, pp. 85–99.

[38] 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.

[39] 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.

[40] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in gcc and llvm," in *ISSTA*, 2016, pp. 294–305.

[41] 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.

[42] V. Jagannath, Q. Luo, and D. Marinov, "Change-aware preemption prioritization," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 133–143.

[43] P. Tonella, P. Avesani, and A. Susi, "Using the case-based ranking methodology for test case prioritization," in *Proceedings of the 22nd International Conference on Software Maintenance*, 2006, pp. 123–133.

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

[45] H. Srikanth, S. Banerjee, L. Williams, and J. Osborne, "Towards the prioritization of system test cases," *Software Testing, Verification and Reliability*, vol. 24, no. 4, pp. 320–337, 2014.

[46] Á. Beszédes, T. Gergely, L. Schrettner, J. Jász, L. Langó, and T. Gyimóthy, "Code coverage-based regression test selection and prioritization in webkit," in *Proceedings of the 2012 28th IEEE International Conference on Software Maintenance*, 2012, pp. 46–55.

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

[48] 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.

[49] Q. Luo, K. Moran, and D. Poshyvanyk, "A large-scale empirical comparison of static and dynamic test case prioritization techniques," in *Proceedings of 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2016, p. to appear.

[50] Q. Gao, J. Li, Y. Xiong, D. Hao, X. Xiao, K. Taneja, L. Zhang, and T. Xie, "High-confidence software evolution," *Science China Information Sciences*, vol. 59(7), pp. 071 101:1–071 101:19, 2016.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] ——, "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.

[62] 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.

[63] 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.

[64] 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.

[65] D. You, Z. Chen, B. Xu, B. Luo, and C. Zhang, "An empirical study on the effectiveness of time-aware test case prioritization techniques," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, 2011, pp. 1451–1456.

[66] T. Y. Chen, H. Leung, and I. Mak, "Adaptive random testing," in *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making*, 2005, pp. 320–329.

[67] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse, "Adaptive random testing: The art of test case diversity," *Journal of Systems and Software*, vol. 83, no. 1, pp. 60–66, 2010.

[68] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 78–88.

[69] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, 2008.

[70] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu, "Dictionary learning based software defect prediction," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 414–423.

[71] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, "Micro interaction metrics for defect prediction," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011, pp. 311–321.

[72] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008, pp. 13–23.

[73] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan, "Cross-project defect prediction using a connectivity-based unsupervised classifier," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 309–320.

[74] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 181–190.

[75] T. T. Nguyen, T. N. Nguyen, and T. M. Phuong, "Topic-based defect prediction," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 932–935.

[76] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, vol. 14, no. 5, pp. 540–578, 2009.

[77] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, 2007.

[78] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 297–308.

[79] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," *Information and Software Technology*, vol. 54, no. 3, pp. 248–256, 2012.