Learning to Accelerate Compiler Testing

Junjie Chen
Key Laboratory of High Confidence Software Technologies (Peking University), MoE
Institute of Software, EECS, Peking University, Beijing, 100871, China
chenjunjie@pku.edu.cn

ABSTRACT

Compilers are one of the most important software infrastructures. Compiler testing is an effective and widely-used way to assure the quality of compilers. While many compiler testing techniques have been proposed to detect compiler bugs, these techniques still suffer from the serious efficiency problem. This is because these techniques need to run a large number of randomly generated test programs on the fly through automated test-generation tools (e.g., Csmith). To accelerate compiler testing, it is desirable to schedule the execution order of the generated test programs so that the test programs that are more likely to trigger compiler bugs are executed earlier. Since different test programs tend to trigger the same compiler bug, the ideal goal of accelerating compiler testing is to execute the test programs triggering different compiler bugs in the beginning. However, such perfect goal is hard to achieve, and thus in this work, we design four steps to approach the ideal goal through learning, in order to largely accelerate compiler testing.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Compilers**;

KEYWORDS

Compiler Testing, Test Prioritization, Machine Learning

ACM Reference Format:
Junjie Chen. 2018. Learning to Accelerate Compiler Testing. In *ICSE '18 Companion: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3183440.3183456

1 INTRODUCTION

Compilers are widely recognized as one of the most important infrastructures, since almost all of software systems depend on it. Buggy compilers may lead to unintended behaviors of developed programs, which may cause software failures or even disasters in safe-critical domains. Also, buggy compilers increase the difficulty of debugging, since developers can hardly determine whether a software failure is caused by the software they are developing or the compilers they are using. Therefore, it is crucial to guarantee the quality of compilers.

Compiler testing is an effective and widely-used way to ensure the correctness of compilers. Many compiler testing techniques have been proposed in the literature to automate compiler testing [5, 7, 8, 10, 11, 16, 19, 21]. These techniques depend on automated test-generation tools (e.g., Csmith [19]) to generate a large number of test programs, which are the inputs of compilers, and compiler bugs are detected by running them. However, based on the existing work [10, 19], compiler testing still suffers from the serious efficiency problem: *compiler testing techniques spend an extremely long period of time to detect a small number of bugs.* For example, Yang et al. [19] spent three years on detecting 325 C compiler bugs, and Le et al. [10] spent eleven months on detecting 147 C compiler bugs. That is, it is necessary to accelerate compiler testing.

During the process of compiler testing, there are a small subset of test programs that are able to trigger compiler bugs [5], and thus compiler testing can be accelerated by running these test programs earlier. That is, test prioritization is likely to be adopted to accelerate compiler testing. However, existing test prioritization approaches can hardly be used to accelerate compiler testing due to the following reasons. On the one hand, the dominant test prioritization approaches depend on coverage information, which is collected through regression testing [20]. However, compiler testing techniques mainly depend on the test programs generated randomly on the fly by automated test-generation tools like Csmith [19]. Therefore, the coverage information of these test programs is not available before testing. On the other hand, there are also some test-input based test prioritization approaches, which depend on only test-input information [9]. However, they also cannot be used to accelerate compiler testing, since they are evaluated that these approaches have bad effectiveness due to extremely long time spent on prioritization [1]. On the whole, traditional test prioritization approaches cannot accelerate compiler testing at all.

To address the serious efficiency problem in compiler testing, in this work we aim to propose an effective approach to scheduling the execution order of test programs such that test programs triggering different compiler bugs are executed earlier. Since different test programs tend to trigger the *same* compiler bug to a large extent, the ideal goal is that test programs triggering *different* compiler bugs are ranked on the top of the scheduled execution order. However, such perfect result is hardly to achieve. In our work, we plan to utilize the idea of learning to help us approach the idea goal.

2 BACKGROUND

To ease understanding of the challenges of accelerating compiler testing, in this section, we introduce the background on compiler testing. To guarantee the quality of compilers, many compiler testing techniques have been proposed. Also, Chen et al. [5] conducted

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5663-3/18/05…$15.00
https://doi.org/10.1145/3183440.3183456

an empirical study to compare three mainstream compiler testing techniques. We introduce the three techniques as follows.

Randomized differential testing (RDT) [14] is one of the most widely-used compiler testing techniques in practice [12, 19]. RDT requires two or more comparable compilers to detect compiler bugs. Here "comparable" means that compilers implement the same specifications. More specifically, given a test program *P* and a set of its inputs *I*, when using a set of comparable compilers to compile *P* and then executing the set of generated executables under *I*, they should produce the same results. If the produced results are different, a compiler bug is detected. When the number of comparable compilers is more than two, they can determine which compiler contains the bug through voting.

Different optimization levels (DOL) [5] is a variant of RDT. It is derived from RDT by replacing different comparable compilers with different optimization levels of a compiler (e.g., -O0, -O1, -Os, -O2, and -O3 in GCC). Given a test program *P* and a set of its inputs *I*, when *P* is compiled under different optimization levels of a compiler, and then executed under *I*, the produced results should be the same. If they are different, a compiler bug is detected.

Equivalence modulo inputs (EMI) is a recently proposed compiler testing technique. It addresses the test oracle problem in compiler testing through comparison between a test program and its variants which are equivalent under a set of test inputs with the test program. Given a test program *P* and a set of its inputs *I*, EMI first generates some equivalent variants with *P* under *I*, and then these variants and *P* are compiled by a compiler and executed under *I*. If the produced results are different, a compiler bug is detected. EMI has three instantiations, i.e., Orion [10], Athena [11], and Hermes [17].

3 RESEARCH PROPOSAL

To address the serious efficiency problem of compiler testing, the ideal goal is to execute test programs triggering different compiler bugs in the beginning. To approach the ideal goal, there are two key challenges as follows.

- How to identify which test programs are able to trigger compiler bugs.
- How to distinguish which test programs trigger different compiler bugs.

Our research proposal mainly focuses on solving the two challenges to accelerate compiler testing. To solve the first challenge, we plan to explore whether there are features from test programs that are helpful to predict which test programs are more likely to trigger compiler bugs to accelerate compiler testing. More specifically, we will first explore the feasibility of using basic features from test-program text, and then explore the feasibility of utilizing the idea of learning on various test-program features. To solve the second challenge, we plan to find a proper approximation to distinguish which test programs trigger different compiler bugs to accelerate compiler testing. Besides, it is also necessary for me to better know various compiler testing techniques before accelerating them. Therefore, we design the following four steps in my dissertation research to approach the ideal goal.

•**The first step aims to explore various compiler testing techniques as the stepping-stone for me to enter the field of compiler testing**. We plan to conduct an empirical study to investigate various compiler testing techniques and analyze factors influencing them. On the one hand, it is helpful for me to better know these compiler testing techniques. On the other hand, it may also further motivate the necessity of accelerating compiler testing.

In the empirical study, we plan to investigate three mainstream compiler testing techniques, including RDT, DOL, and EMI. More specifically, we plan to use Csmith [19] to generate 100,000 test programs to test GCC and LLVM using the three compiler testing techniques. We will analyze these techniques from various aspects, e.g., effectiveness, cost-effectiveness, and substitutability.

• **The second step aims to explore whether compiler testing will be accelerated by considering basic information from test-program text to solve the first challenge**. As mentioned in Section 1, coverage-based test prioritization can hardly be applied to accelerate compiler testing, and thus we plan to schedule the execution order of test programs based on only test-program information. Fortunately, the inputs of compilers are test programs, which contain a great deal of information that reflects compiler bugs to some extent, and thus it is possible to extract bug-relevant tokens from test programs by regarding them as text, similar to existing input-based test prioritization [9]. That is, we will schedule the execution order of test programs based on these bug-relevant tokens from test-program text in this step.

In the evaluation, we plan to compare our approach with the baseline approach, i.e., random order, which randomly determines the execution order of test programs, to investigate the acceleration effectiveness of our approach. In particular, the subjects we plan to use are two popular open-source C compilers, i.e., GCC and LLVM, and the compiler testing techniques to be accelerated are two mainstream techniques, i.e., DOL and EMI.

• **The third step aims to explore the direction of accelerating compiler testing where we utilize the idea of learning to systematically use various test-program information to further solve the first challenge**. According to the results of the second step, if such basic test-program information can accelerate compiler testing, it provides us a promising direction to fasten compiler bug detection. That is, we can systematically learn the relationship between the bug-revealing characteristics of test programs and the discovery of compiler bugs to accelerate compiler testing. In this step, we plan to systematically identify a set of bug-revealing characteristics (e.g., program structures and element usage) from test programs by analyzing test programs triggering historical compiler bugs. Based on such information, we plan to learn a prediction model to predict the bug-revealing probability of each new test program. Furthermore, it is also important to consider the execution time of each test program in the practical acceleration, since test programs with higher bug-revealing probability but extremely long execution time may decelerate compiler testing. The idea of learning also provides an opportunity to predict the execution time of each test program. That is, we will schedule the execution order of test programs by considering the bug-revealing probability and execution time of each test program.

To investigate the acceleration effectiveness of the proposed approach, we will compare it with both the baseline approach (i.e., random order) and the approach proposed in the second step. In particular, we will evaluate the learning capability of our approach on cross-version and cross-compiler scenarios. That is, we will

investigate whether our proposed approach can further accelerate compiler testing regardless of the models trained based on a previous version of the compiler under test or another compiler.
• **The fourth step aims to explore the degree to which distinguishing different test programs triggering different compiler bugs further facilitates compiler testing acceleration**. During the second and third steps, we explore the potential of accelerating compiler testing by just solving the first challenge. However, both of them ignore the important problem in the second challenge, where different test programs tend to trigger the same compiler bug to a large extent. The neglect of this problem may largely discount the acceleration effectiveness of these approaches, while these approach may have achieve good accelerate effectiveness. In this step, we will propose an approach to also solving the second challenge for further acceleration of compiler testing.

It is impossible for us to know which test programs trigger different compiler bugs before executing them. Therefore, the only possible direction is to find a proper approximation to distinguish them. Test coverage is one of the most widely-used methods to measure the effectiveness of tests, and has been widely adopted by researchers and practitioners to improve software testing process, such as test-suite reduction [3]. That is, test coverage may be a proper approximation to help distinguish which test programs trigger different compiler bugs. Intuitively, if test programs have quite different coverage information, they are more likely to trigger different compiler bugs. Unfortunately, test coverage tends to be collected by instrumenting programs at a series of locations (e.g., at the entry and exit points of methods) and then executing the instrumented programs, which cannot be applied to compilers because most test programs of compilers are generated randomly on the fly by automated test-generation tool like Csmith [19]. That is, test coverage of compilers cannot be acquired before testing.

In this step, we plan to propose a method to statically predict test coverage information through learning without dynamic execution, based on the historical coverage information and test program data. According to the predicted coverage, we will cluster test programs into different groups, and test programs in different groups are more likely to trigger different compiler bugs. Finally, we will schedule the execution order of test programs based on the clustering results and their bug-revealing probabilities per unit time predicted by the approach proposed in the third step. That is, this approach is to accelerate compiler testing by solving both of the two challenges.

In the evaluation, we will compare our proposed approach with the baseline approach (i.e., random order) and the approach proposed in the third step to investigate whether our proposed approach can further accelerate DOL and EMI on GCC and LLVM. Besides, we also investigate the accuracy of our proposed approach on predicting coverage of compilers. In particular, we plan to compare our prediction method with the baseline approach (i.e., random guess), which randomly guesses the covered code regions of compiler by each test program.

4 RESEARCH PROGRESS

In this section, we first introduce the finished work of my dissertation research in Section 4.1, and then present the remaining work and plan in Section 4.2.

4.1 Finished Work

Before accelerating compiler testing, we first conducted an empirical study to explore various compiler testing techniques [5]. Then we completed the approaches proposed in the second and third steps of my research proposal, including TB−G [2] and LET [1].
•**Empirical Investigation on Compiler Testing Techniques**. As the first work of my dissertation research, we conduct an empirical study to explore three mainstream compiler testing techniques, including RDT, DOL, and EMI. According to our empirical study, we find that any of the three mainstream compiler testing techniques just detected no more than 20 bugs (on GCC or LLVM) by running 100,000 randomly generated test programs through Csmith. That further confirms the serious efficiency problem of compiler testing. That also further motivates us to accelerate compiler testing by running those test programs triggering compiler bugs earlier. This work has been published in ICSE 2016 [5].
•**TB−G Acceleration Approach**. To accelerate compiler testing, we proposed the first approach called TB−G by solving the first challenge approaching our ideal goal. It regards test programs as text, and then extracts the tokens reflecting bug-relevant characteristics from text so as to transform each test program into a text-vector. The extracted characteristics contain statement characteristics, type and modifier characteristics, and operator characteristics. After acquiring text-vectors, TB−G normalizes the values of elements in text-vectors into an interval between zero and one. Finally, TB−G computes the distance between normalized text-vectors and the origin vector, and then prioritizes test programs based on the descending order of their distances. Our experimental results demonstrate that TB−G significantly accelerates compiler testing on GCC and LLVM. The success of the second step demonstrates the potential of prioritizing test programs based on only test-program information. This work has been published in ICST 2016 [2].
•**LET Acceleration Approach**. LET is proposed recently to accelerate compiler testing, which further better solves the first challenge. It accelerates compiler testing based on the historical bug information through learning. LET contains two processes: learning process and scheduling process. In the learning process, LET first identifies bug-revealing features from test programs. In particular, the features have two types, i.e., existence features and usage features. The former is concerned with whether certain types of elements (e.g., expressions and operators) exist in the target program, while the latter is concerned with how the elements in a program are used. Based on these features, LET trains a capability model, which is used to predict bug-revealing probability of each new test program. Moreover, LET trains a time model, which is used to predict execution time of each new test program. In the scheduling process, LET also extracts these features from new test programs. Based on the two trained models, LET predicts bug-revealing probability and execution time of each new test program, and then ranks new test programs based on the descending order of their bug-revealing probabilities per unit time. Our experimental results show that LET significantly accelerates compiler testing regardless of on the cross-version or cross-compiler scenarios. In particular, we also compared LET with TB−G, and our results show that LET is more effective and stable than TB−G for accelerating compiler testing. The success of the third step demonstrates the ability of learning on

accelerating compiler testing by learning the relationship between bug-revealing characteristics of test programs and the discovery of compiler bugs. This work has been published in ICSE 2017 [1].

4.2 Remaining Work

Our remaining work is our fourth step to approach the ideal goal. The above-mentioned proposed approaches (i.e., TB−G and LET) indeed accelerate compiler testing, while they do not consider the important problem in compiler testing where different test programs tend to trigger the same compiler bug to a large extent. Such neglect may largely discount the acceleration effectiveness of these approaches. Therefore, our novel approach described in the fourth step is promising to further accelerate compiler testing by solving both of the two challenges.

Our novel approach contains three parts. The first part is to predict test coverage of compilers. This part also utilizes the idea of learning, and the identified features in LET is helpful for us to build the relationship between test-program data and historical coverage information. Furthermore, deep learning may be also an effective direction to predict coverage, since it is easy for us to collect enough training data by randomly generating a large number of test programs. The second part is to cluster test programs into different groups based on the predicted test coverage. Test programs in different groups are more likely to trigger different compiler bugs. In this part, we will learn the knowledge about clustering algorithms, which is different with our used learning algorithms in LET. The third part is to schedule the execution order of test programs. Since we have implemented LET to predict bug-revealing probability per unit time of each test program, we need to design an effective and efficient ranking algorithm considering clustering results. In my plan, the implementation of this approach and all the evaluation to investigate its effectiveness, will be completed before the ending of summer in 2018.

5 RELATED WORK

Our work is related to compiler testing. Based on the definition of tests [4], we classify the existing work on compiler testing into test-program generation and test-oracle definition.

The former is to investigate how to generate test programs for a compiler following some specification (e.g., C99 for C program). In general, random generation is the mainstream technique [6, 13, 15, 19]. For example, Yang et al. [19] proposed a tool Csmith for C compilers, which randomly generates C programs. Based on CSmith, Lidbury et al. [13] developed CLsmith to test OpenCL compilers.

To address the test-oracle problem of complex software systems, including compilers, McKeeman et al. [14] coined the term of differential testing, which detects bugs through comparing at least two comparable compilers. Recently, Le et al. [10, 11, 17] proposed to generate some equivalent variants for each original C program and determine whether a compiler has bugs by comparing the results produced by the original program and its variants. Tao et al. [18] proposed to construct metamorphic relations in compiler testing.

Different with them, our work addresses another important problem in compiler testing, i.e., the test efficiency problem. That is, our work targets at accelerating compiler testing.

6 CONCLUSION

The ideal goal of accelerating compiler testing is to execute test programs triggering different compiler bugs in the beginning. However, such perfect result is hard to achieve. In my dissertation research, we design four steps to approach the ideal goal by finally solving two challenges through learning. The four steps include conducting an empirical study to investigate various compiler testing techniques, proposing two acceleration approaches by solving the first challenges based on basic features from test programs and learning on various features respectively, and proposing a final acceleration approach by solving both of the two challenges.

7 ACKNOWLEDGMENT

Junjie is advised by Prof. Lu Zhang, Prof. Dan Hao, and Prof. Yingfei Xiong. This work is supported by the National Natural Science Foundation of China under Grant No. 61672047 and 61522201.

REFERENCES

[1] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to prioritize test programs for compiler testing. In *ICSE*. 700–711.

[2] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. Test case prioritization for compilers: A text-vector based approach. In *ICST*. 266–277.

[3] Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, and Bing Xie. 2017. How Do Assertions Impact Coverage-Based Test-Suite Reduction?. In *ICST*. 418–423.

[4] Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, Bing Xie, and Hong Mei. 2016. Supporting oracle construction via static analysis. In *ASE*. 178–189.

[5] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *ICSE*. 180–190.

[6] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming Compiler Fuzzers. In *PLDI*. 197–208.

[7] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *PACMPL* 1, OOPSLA (2017), 93:1–93:29.

[8] Alastair F Donaldson and Andrei Lascu. 2016. Metamorphic testing for (graphics) compilers. In *Proceedings of the 1st International Workshop on Metamorphic Testing*. 44–47.

[9] Bo Jiang and W. K. Chan. 2015. Input-based adaptive randomized test case prioritization: A local beam search approach. *JSS* 105, C (2015), 91–106.

[10] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *PLDI*. 25.

[11] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *OOPSLA*. 386–399.

[12] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Randomized stress-testing of link-time optimizers. In *ISSTA*. 327–337.

[13] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core Compiler Fuzzing. In *PLDI*. 65–76.

[14] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[15] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *PLDI*, Vol. 47. 335–346.

[16] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and analyzing compiler warning defects. In *ICSE*. 203–213.

[17] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *OOPSLA*. 849–863.

[18] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. 2010. An automatic testing approach for compiler based on metamorphic testing technique. In *APSEC*. 270–279.

[19] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI*. 283–294.

[20] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. 2013. Bridging the gap between the total and additional test-case prioritization strategies. In *ICSE*. 192–201.

[21] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *PLDI*. 347–361.