2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-6654-3786-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/SANER53432.2022.00065

Isolating Compiler Optimization Faults via Differentiating Finer-grained Options

Jing Yang*†, Yibiao Yang‡§, Maolin Sun*†, Ming Wen*†, Yuming Zhou‡§, Hai Jin†¶
*Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering
Huazhong University of Science and Technology, Wuhan, China
†National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab
Huazhong University of Science and Technology, Wuhan, China
‡State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
§Department of Computer Science and Technology, Nanjing University, China
¶Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST, Wuhan, China
{yjing, merlinsun, mwenaa, hjin}@hust.edu.cn, {yangyibiao, zhouyuming}@nju.edu.cn

***Abstract*—Code optimization is an essential feature for compilers and almost all software products are released by compiler optimizations. Consequently, bugs in code optimization will inevitably cast significant impact on the correctness of software systems. Locating optimization bugs in compilers is challenging as compilers typically support a large amount of optimization configurations. Although prior studies have proposed to locate compiler bugs via generating witness test programs, they are still time-consuming and not effective enough. To address such limitations, we propose an automatic bug localization approach, `ODFL`, for locating compiler optimization bugs via differentiating finer-grained options in this study. Specifically, we first disable the fine-grained options that are enabled by default under the bug-triggering optimization levels independently to obtain bug-free and bug-related fine-grained options. We then configure several effective passing and failing optimization sequences based on such fine-grained options to obtain multiple failing and passing compiler coverage. Finally, such generated coverage information can be utilized via Spectrum-Based Fault Localization formulae to rank the suspicious compiler files. We run `ODFL` on 60 buggy GCC compilers from an existing benchmark. The experimental results show that `ODFL` significantly outperforms the state-of-the-art compiler bug isolation approach RecBi in terms of all the evaluated metrics, demonstrating the effectiveness of `ODFL`. In addition, `ODFL` is much more efficient than RecBi as it can save more than 88% of the time for locating bugs on average.**

***Index Terms*—Compiler, Bug Isolation, Fault Localization, Finer-grained, Optimization Option**

I. Introduction

Compilers are the most fundamental infrastructures in software development. Bugs hidden in compilers will inevitably cast significant impacts on the correctness of the compiled software systems. The correctness of compilers is thus of critical importance. However, being complex software systems, compilers themselves are prone to errors. Consequently, many compiler bugs are being reported every day. Among those bugs, compiler optimization bugs account for the largest proportion [1]. Thus, one of the most important tasks for developers is to locate and fix optimization bugs in compilers. Nevertheless, locating compiler bugs is challenging as compilers are one of the largest software systems. Driven by this, it is of significant importance to advance the techniques for locating bugs of compiler optimizations.

Recently, Chen et al. proposed DiWi [2] and RecBi [3] to facilitate compiler bug isolation. They introduced a novel concept of witness programs, which are mutated from the bug-triggering test programs of the compiler. A mutated test program is considered as a witness program when it does not trigger any bugs of the compiler. The witness test programs can be viewed as the passing test programs in the **S**pectrum-**B**ased **F**ault **L**ocalization (SBFL) techniques [4], [5]. Specifically, DiWi utilizes traditional local mutation operators and Metropolis-Hastings (MH) [6] algorithm to find effective witness programs, while RecBi mutates the structure of programs and utilizes the reinforcement learning [7] algorithm to select programs. Then, both of them leverage the SBFL formula, Ochiai [8], to locate compiler bugs by comparing the coverage information between the bug-triggering program and the generated passing programs.

Although these witness test programs based approaches are effective for compiler bug localization, they still encounter inevitable limitations. First, the goal of the existing approaches is to generate witness test program (i.e., passing test program) which will not produce multiple failing coverage of compilers. Second, generating passing test programs is time-consuming as it is hard to obtain a witness test program without a large amount of attempts in program mutation. Third, the localization results are not reliable due to the randomness of the generated programs. To mitigate this influence, the experiment must be repeated multiple times to obtain relatively reliable results. For example, in GCC bug 58570, we found that the position of the buggy file located by RecBi floated from 21st to 47th among five different experiments. Forth, diverse mutated programs and coverage information is hard to obtained for effective localization when the given failing test program has a simple program structure. Besides, there is little difference between the results of RecBi and DiWi (see more details in Section V), indicating that due to the limited diversity of the coverage information of the mutation

978-1-6654-3786-8/22/$31.00 ©2022 IEEE
DOI 10.1109/SANER53432.2022.00065

Authorized licensed use limited to: Nanjing University. Downloaded on February 23,2023 at 03:22:40 UTC from IEEE Xplore. Restrictions apply.

programs, adapting different sampling strategies does not make substantial contributions.

Approach. In this paper, we therefore propose a simple yet efficient and effective fault localization approach for compilers via differentiating finer-grained options. Here, we only locate optimization bugs in compilers since compiler bugs tend to occur in the components of compiler optimizations. Moreover, all bugs in the benchmark released by RecBi are optimization bugs. Meanwhile, compilers (e.g., GCC [9] and LLVM [10]) support multiple fine-grained optimization options under coarse-grained optimization levels and each option can be enabled or disabled manually. Take GCC for example, the finer-grained optimization option *-foptimize-strlen* optimizes several standard C string functions (such as strcpy and strchr), which is enabled by default at the coarser-grained optimization levels of O2 and O3. Almost all optimization bugs in compilers are reported at the coarser-grained optimizations. However, bugs are often caused by one or several specific finer-grained optimizations. Moreover, we also observed that the compiler execution results can be flipped (from failing to passing) or keep failing, via disabling the finer-grained optimization option that is enabled by default under the bug-triggering coarser-grained optimization level. That means we can obtain the failing and passing coverage information of compilers by differentiating finer-grained optimization options. Overall, the goal of differentiating finer-grained options is to configure effective passing and failing finer-grained optimization sequences, which can be utilized to obtain the compiler coverage information for fault localization.

Based on this intuition, to speed up the configuration process, we present `ODFL` in this study. Specifically, `ODFL` first disables each of the finer-grained optimization options under the bug-triggering coarser-grained optimization levels one after another. If the compilation result flipped when the finer-grained optimization is disabled, it is considered as a bug-related option by `ODFL`. Otherwise, it is considered as a bug-free option. `ODFL` then adds bug-related and bug-free options into the initial finer-grained option sets. Second, `ODFL` configures effective failing and passing optimization options based on the initial option sets. We have two goals in the process of configuration: For one thing, each failing optimization configuration enables as few optimizations as possible, to reduce the suspicion of innocent complier lines and files. For another thing, each passing optimization configuration should be similar to the failing optimization configuration, to ensure their similar execution paths. Finally, similar to the existing work [2], [3], `ODFL` calculates all the suspicious scores of the suspicious files according to the SBFL formula based on the passing coverage information and the failing coverage information. In general, the passing and failing coverage information we collected by differentiating finer-grained optimization options under bug-triggering coarser-grained optimization levels, has been demonstrated to be more effective and efficient than the state-of-the-art compiler fault localization technique.

We utilized 60 real GCC [9] bugs from the released benchmark constructed in RecBi to evaluate the effectiveness of `ODFL`. The experimental results demonstrate that `ODFL` requires less time to locate compiler bugs and significantly outperforms the existing compiler fault localization approaches (i.e., DiWi [2] and RecBi [3]). Overall, within Top-1, Top-5, Top-10, and Top-20 files, `ODFL` isolates 15, 33, 48, 54 compiler bugs (out of 60 GCC bugs) respectively. Compared with RecBi, the state-of-the-art fault localization approach of compilers, `ODFL` isolates 87.50%, 57.14%, 41.18%, and 22.73% more bugs within Top-1, Top-5, Top-10, and Top-20 files. Besides, we investigated the contributions of both the passing and failing coverage information collected by differentiating finer-grained optimization options. Our results show that both of them contribute significantly to locating optimization bugs of compilers. Furthermore, we evaluated whether applying different SBFL formulae on `ODFL` affects its performance. Our experimental results demonstrate that `ODFL` also achieves significant performance using other SBFL formulae such as DStar [11], Tarantula [12], Op2 [13], Ochiai2 [13], and Barinel [4].

Contributions. We make the following main contributions:

- We propose `ODFL`, the first fault localization approach for compilers based on differentiating optimization options. It improves the effectiveness and efficiency of existing approaches significantly in locating compiler bugs.
- We propose diverse failing coverage information for calculating the suspicious value for code elements of compilers, which has enriched the conventional coverage information, thus achieving more effective localization.
- We implemented our approach as a practical fault localization tool of compliers, based on the instrumentation tool Gcov [14], which can be easily extended.
- We conducted extensive empirical experiments on 60 bugs of the GCC compiler. The results demonstrate the effectiveness and efficiency of `ODFL`, as well as the contribution of each major component of `ODFL`.

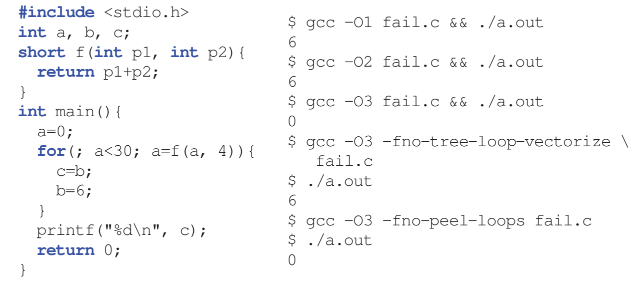
II. Background & Motivation

We first introduce the SBFL technique utilized in our study. Then, we use a concrete example to illustrate our motivation of configuring compiler optimization options for the localization of compiler bugs.

A. Spectrum-Based Fault Localization

SBFL is designed to locate and rank suspicious code elements based on the execution paths of passing and failing test cases. Prior studies have emphasized that SBFL techniques require sufficient failing test cases to accurately reveal a fault [15]. Unfortunately, the state-of-the-art fault localization approaches of compliers only utilized one failing test case, which is the given failing test program. This is because, it is challenging to generate failing test programs that trigger the same bug as the given test program, and thus multiple failing test programs cannot be easily obtained through program mutations. In this study, we observe that the novel perspective, which is configuring finer-grained optimization options, is

Authorized licensed use limited to: Nanjing University. Downloaded on February 23,2023 at 03:22:40 UTC from IEEE Xplore. Restrictions apply.



```
#include <stdio.h>
int a, b, c;
short f(int p1, int p2){
  return p1+p2;
}
int main(){
  a=0;
  for(; a<30; a=f(a, 4)){
    c=b;
    b=6;
  }
  printf("%d\n", c);
  return 0;
}
```

(a) Failing test program

```
$ gcc -O1 fail.c && ./a.out
6
$ gcc -O2 fail.c && ./a.out
6
$ gcc -O3 fail.c && ./a.out
0
$ gcc -O3 -fno-tree-loop-vectorize \
  fail.c
$ ./a.out
6
$ gcc -O3 -fno-peel-loops fail.c
$ ./a.out
0
```

(b) Compile configuration

Fig. 1. GCC bug #71439. The failing program was compiled correctly under the O2 and O1 optimization option, and incorrectly compiled under the O3 optimization option. When disabling the fine-grained optimization option "-ftree-loop-vectorize" under O3, the result filpped (from failing to passing). However, after disabling the other option "-fpeel-loops", the bug still occurred.

capable of generating rich information of both passing and failing coverage of compilers.

B. Compiler Optimization Option Configuration

Optimization is one of the most fundamental components of compilers, and a compiler usually supports hundreds of finer-grained optimization options under a certain coarse-grained optimization option for code optimization. Each of the finer-grained optimization options can be enabled or disabled independently. For instance, we can disable the optimization option "*-fpeel-loops*", a finer-grained option enabled default by the coarse-grained optimization option "-O3" in GCC, via "*-fno-peel-loops*". To investigate the effects of such finer-grained options on compilers bugs, we disabled the finer-grained optimization option that is enabled at the bug-triggering optimization level, and then used such configurations to compile the given failing test programs. We further examined the execution results under such configurations, and found that the compiler execution results can either be flipped (from failing to passing) or keep failing. Based on such observations, we introduce the following definitions:

- **Bug-related Option:** A finer-grained optimization option is *bug-related* if the compilation result ***flips*** when the option is disabled at the bug-triggering optimization level.
- **Bug-free Option:** A finer-grained optimization option is *bug-free* if the compilation result ***keeps failing*** when the option is disabled at the bug-triggering optimization level.
- **Failing Optimization Configuration:** A failing optimization configuration is a combination of multiple finer-grained optimization options (either bug-related or bug-free) that trigger the bug for the given test program.
- **Passing Optimization Configuration:** A passing optimization configuration is a combination of multiple finer-grained optimization options (either bug-related or bug-free) that does not trigger the bug for the given test program.

Example. In this section, we use a concrete example in Figure 1 to illustrate the motivation of utilizing optimization option configuration in fault localization of compilers. This figure shows the real bug of GCC #71439, where the left is the reported bug-triggering test program and the right is the corresponding compiler option configurations. From Figure 1, we can find that the output of the test program under the optimization levels "-O1" and "-O2" is 6. However, under the optimization leve "-O3", the output is a different value 0, thus triggering a bug in the compiler GCC. In addition, when the finer-grained optimization option "*-ftree-loop-vectorize*" (which is turned on by default under "-O3") is disabled by using the "*-fno-tree-loop-vectorize*" option, the output of the test program turns to be correct as the value is changed to 6. According to our above definitions, "*-ftree-loop-vectorize*"' is a ***bug-related option*** and "*-O3 -fno-tree-loop-vectorize*" is a ***passing optimization configuration***. In addition, when disabling "*-fpeel-loops*" via "*-fno-peel-loops*", the output of the test program is still incorrect as the value remains unchanged. Accordingly, "*-fpeel-loops*" is a ***bug-free option*** and "*-O3 -fno-peel-loops*" is a ***failing optimization configuration***.

With the identified finer-grained bug-related options and bug-free options, we are able to obtain multiple passing and failing optimization configurations by enabling or disabling some of those fine-grained options from the coarse-grained bug-triggering optimization level. Meanwhile, we can obtain the coverage information of compilers for each of the passing or failing optimization configurations. We further analyze whether more effective failing and passing coverage information can be obtained by configuring optimization options. In practice, we discovered that disabling more optimization options under the bug-triggering optimization level to construct a failing optimization configuration helps to locate compiler bugs (Section III explains it in detail). Also, the passing optimization configurations that are similar to the failing one are required to follow SBFL effectively. However, since hundreds of options are enabled under different optimization levels, the search space is huge with brute force enumeration. To speed up the construction process, we configure optimization sequences within the bug-related and bug-free options. On the one hand, bug-related options are finer-grained optimizations that trigger the bug, and thus using them to construct optimization configuration can locate bugs more accurately. On the other hand, considering the bugs may be triggered at the combination of multiple options, we add bug-free options to the initial construction space as well. Overall, the optimization configuration consists of a sequence of finer-grained options and the corresponding bug-triggering optimization level, and our goal is to construct effective passing and failing optimization configurations based on the bug-related and bug-free options.

Therefore, ODFL designs the configuring optimization option mechanism to produce both failing and passing coverage information, which improves the effectiveness and efficiency of compiler fault localization.

III. Compiler Optimization & Compiler Coverage

In this section, we first discuss the statistical information about the number of bug-related and bug-free options. We then focus on the effect of configuring diverse numbers of compiler optimizations on compiler coverage, which is essential for effective localization of compiler bugs via utilizing optimization configurations.

Authorized licensed use limited to: Nanjing University. Downloaded on February 23,2023 at 03:22:40 UTC from IEEE Xplore. Restrictions apply.

TABLE I

THE STATISTICS OF THE NUMBER OF BUG-RELATED OPTIONS AND THE BUG-FREE OPTIONS FOR THE RELEASED BUGS IN THE BENCHMARK.

Bug-triggering level	Type	Min	Max	Median	Mean
Lowest	Bug-related	1	14	5	5.58
Lowest	Bug-free	67	135	110	107.98

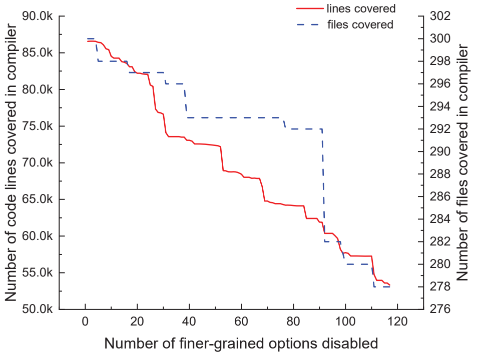


Fig. 2. The effect of optimization quantity on code coverage of compilers.

A. Compiler Optimization Option

As described in Section II-B, we construct optimization configurations with the set of bug-related and bug-free options to enhance the effectiveness and efficiency of ODFL. To achieve such a goal, it is necessary to obtain moderate bug-free and bug-related options for configuring the optimization sequences. Therefore, we discuss the statistical information about the number of these two types of options as follows.

First, in most cases, when a compiler optimization bug occurs at a low optimization level, the same bug will also occur at the higher optimization levels (e.g., -O2 and -O3 both are bug-triggering optimization levels in GCC bug #58570 [16]). Therefore, in our study, we only take the lowest bug-triggering optimization level into consideration since higher optimization levels enable more bug-free optimizations, which hinders the fault localization of compilers. Second, we analyzed the number of the bug-related and the bug-free options for each buggy compiler in the benchmark under the lowest bug-triggering optimization level. Table I presents the statistics of the number of the bug-related and bug-free options for all the bugs in the benchmark. Rows "Lowest" in this table mean we count the number of options under the lowest bug-triggering optimization levels. To summarize, as shown in Table I, we found that there are more bug-free options than bug-related options in buggy compilers on average. Specifically, each lowest bug-triggering optimization level owns about 5 bug-related options and 107 bug-free options. Consequently, by disabling a large number of bug-free options, we can eliminate many innocent optimizations from being suspected. Therefore, we have adequate options to configure optimization sequences for effective fault localization of compilers.

B. Effect of the Optimization Quantity on Compiler Coverage

An important aspect of the effectiveness of optimization option configuring based method is to search for failing optimization configurations that enable few optimizations. This is because more innocent optimizations can be eliminated from being suspected in this way, which reduces suspicious source code of compilers and has positive impact on fault localization of compilers. Under this intuition, we analyzed the number of compiler lines and files covered under failing optimization configurations consisting of different number of optimizations, thus facilitating failing optimization configuration selection in our research. In the following, we present at length the effect of different numbers of optimizations used to compile programs on the coverage information of compilers.

Based on the intuition that disabling more optimizations allows the buggy compiler to have fewer code elements (e.g., lines and files) to be covered. We therefore incrementally disable fine-grained optimization options under the bug-triggering optimization level of the compiler. Figure 2 shows the effect of using different numbers of optimizations on the number of lines and files covered in compilers, using GCC revision 199531 as an example. The x-axis of the figure is the number of fine-grained optimizations that were turned off, and the number ranges from 1 to 120. The left Y-axis is the number of lines covered in the compiler, reducing from about 86,000 to about 53,000. The right y-axis is the number of files covered in the compiler, which decreases from about 300 files to about 278 lines. From this figure, we observed that the more optimizations are disabled, the fewer code elements are covered in compilers. That means, disabling more optimization options can simplify the coverage information of compilers to a great extent. Furthermore, applying this observation to obtaining failing coverage information helps to eliminate many innocent code elements from suspicious ones.

From the above investigation, we make the following observation: failing optimization configurations with as few optimizations enabled as possible contribute to simplifying suspicious coverage of compilers.

IV. APPROACH

Following the observation in Section III, we design a novel fault localization approach of compilers via differentiating fine-grained optimization options, named ODFL (**O**ption **D**ifferentiating for **F**ault **L**ocalization). In the following, we introduce the overview of ODFL's architecture in Section IV-A and present our approach in detail in Section IV-B. Section IV-C presents the SBFL formula applied in ODFL.

A. Overview

In ODFL, we deliberately search several effective failing optimization configurations and passing optimization configurations, and then compare their compiler coverage similar to SBFL to locate compiler bugs. Thus, there are two key issues to the success of ODFL. First, from Section III, we know that using fewer optimizations to compile programs makes the code coverage of compilers reduced. To reduce the number of innocent lines being executed in the buggy compilers, the failing optimization configurations ODFL utilized should enable as few optimizations as possible. Meanwhile, the passing coverage sharing similar execution traces with the failing

Authorized licensed use limited to: Nanjing University. Downloaded on February 23,2023 at 03:22:40 UTC from IEEE Xplore. Restrictions apply.

coverage of compilers is also helpful to eliminate innocent files from being suspected. Overall, to help search several effective failing and passing optimization configurations for locating compiler bugs, we should achieve the following two goals.

Goal 1. Failing optimization configurations should enable as few finer-grained options as possible to reduce suspicious source code of compilers.

Goal 2. Passing optimization configurations should enable as similar finer-grained options as possible to the failing ones to differentiate innocent files from other suspicious ones.

However, since the search space of the optimization options is extremely huge (about hundreds of option enabled under a optimization level), efficiently searching such several configurations achieving the two above goals is challenging. To tackle such a challenge, `ODFL` disables all bug-free options and enables all bug-related options to initialize a optimization configuration. This is because enabling all bug-related optimizations can trigger bugs with a high probability, and disabling all bug-free options reduces the use of innocent optimizations. That is, with the help of the initial optimization configuration, `ODFL` is able to quickly find the failing optimization configurations that uses few optimizations. Then, `ODFL` utilizes the searched failing optimization configurations to guide the selection of effective passing optimization configurations.

B. Optimization Option Configuration

During the process of searching for effective failing and passing optimization configurations, `ODFL` first initializes a optimization option configuration with bug-related and bug-free options (described in Section IV-B1), and then selects configurations according to search rules based on the initial configuration (described in Section IV-B2).

1) Optimization Configuration Initialization: There are hundreds of the fine-grained optimization options under each coarse-grained optimization level in compilers, making the process of differentiating optimization configurations via exhaustive search impossible. In this case, bug-related and bug-free options serve as basic elements in the initial search space with the aim to reduce search cost. However, multiple options might negatively influence each other in some cases, and the compiler bug might only occur while certain options are enabled collectively. For this reason, we consider both bug-related and bug-free options as the configuration elements in our search space even though bug-related ones are more likely to trigger the bug. Specifically, `ODFL` initially constructs a optimization option configuration via disabling all bug-free options and enabling all bug-related options. This is because it can use fewer optimizations to trigger bugs with the highest probability in this way. That means, to reduce the search cost, `ODFL` disables all bug-free options and enables all bug-related options to configure an initial optimization sequences.

We denote the optimization options under the lowest bug-triggering level in the search space as $F = \{f_{br_1}, ..., f_{br_n}, f_{bf_1}, ..., f_{bf_m}\}$, where f_{br_i} (bug-related option) and f_{bf_i} (bug-free option) can be set to either 0 or 1 (0/1 refers to disabling/enabling the option respectively). Besides, n and m refer to the number of the bug-related options and bug-free options respectively. Since each option in the search space is the potential bug-triggering option, the size of the optimization configuration space is $\sum_{i=1}^{n+m} C_{n+m}^{i} = 2^{n+m}$ ($n+m$ is the size of the search space). Due to the huge search space and limited computational resources, we cannot search all optimization configurations exhaustively. With the aim to search for the failing configuration with few optimizations, `ODFL` initially disables all bug-free options and enables all bug-related options, guided by the intuition that bug-related options are more related to the compiler bug. Therefore, the initial search set is $F = \{f_{br_1}, ..., f_{br_n}, f_{bf_1}, ..., f_{bf_m}\}$, where f_{br_i} is set to 0 and f_{bf_i} is set to 1. Then, `ODFL` searches for failing and passing optimization option configurations via disabling the enabled options or enabling the disabled options, we introduce this process in Section IV-B2 in detail.

2) Optimization Configuration Selection: An important aspect to the success of the optimization configuration differentiation based approach is how to select effective failing and passing optimization configurations. `ODFL` follows the principle that searching for the failing configuration first, and then identifying the passing configurations that enable similar optimizations to the failing one.

According to the analysis in Section IV-A, the selection of the passing and failing configurations should achieve both ***Goal 1*** and ***Goal 2***. To achieve ***Goal 1***, `ODFL` first initializes the configuration with disabled bug-free options and enabled bug-related options. Next, `ODFL` operates on the options in the search set based on the initial configuration. Specifically, each optimization configuration produces two types of compilation results (passing and failing), correspondingly, the search process can be summarized into three operations: ① If the compilation result is failing under the current optimization configuration, `ODFL` disables those enabled options with combinatorial search for the next configuration, until sufficient passing configurations are found (achieving the ***Goal 2***). When one or more failing configurations are found during the process, the new ones are added to the failing configurations since they contain fewer enabled options, and continue to execute this step until the termination condition is reached. ② If the compilation result is passing under the current optimization configuration and the failing configuration has been found before, `ODFL` performs the same operation as the first one, i.e., disables those enabled options with combinatorial search for the next configuration. ③ If the compilation result is passing under the current optimization configuration and the failing configuration has never been found yet, `ODFL` enables those disabled options with combinatorial search for the next configuration until it finds the first failing optimization configuration. `ODFL` then performs the first operation to search passing and failing optimization configurations. Here, effective passing and failing configurations can be searched according to the above three operations via achieving two search goals in Section IV-A.

Since `ODFL` either operates in the enabled option set or in the disabled option set, the search space is reduced from 2^{n+m}

Authorized licensed use limited to: Nanjing University. Downloaded on February 23,2023 at 03:22:40 UTC from IEEE Xplore. Restrictions apply.

to 2^n or 2^m. In most cases, the i in $\sum_{i=1} C_n^i$ or $\sum_{i=1} C_m^i$ is no more than 4 thanks to the initial search space we set.

In this way, the passing optimization configurations share similar coverage with the failing optimization configurations. Moreover, to increase the diversity of the passing coverage information, `ODFL` disables the bug-related options one after another to obtain more passing coverage of the compiler. `ODFL` also utilizes the passing code coverage of compilers via using passing optimization levels to compile programs. The added passing coverage differs from the coverage obtained based on searched passing optimization configurations significantly, since the optimizations they used are quite different.

Algorithm 1: Differentiating Optimization Option

```
   Input: $O_{level}$: The lowest optimization level; $F$: Initial option search space.
   Output: $C_{fail}$: The set of failing optimization configurations; $C_{pass}$: The
           set of passing optimization configurations.
1  $F \leftarrow [1_1, ..., 1_n, 0_1, ..., 0_m]_{n+m}$
   /* initialize optimization configuration */
2  $c_{tmp} \leftarrow$ OptionSequence($F, O_{level}$)
3  while no termination criterion met do
4      if $c_{tmp}$ is failing then
5          $C_{fail} \leftarrow C_{fail} \cup \{c_{tmp}\}$
6          for $i \leftarrow 1$ to $n$ do
               /* combine the enabled options and disable
                  them */
7              $\mathbb{F} \leftarrow$CombineAndDisable($F_1, i$)
8              for $F'$ in $\mathbb{F}$ do
9                  $c_{tmp} \leftarrow$ OptionSequence($F', O_{level}$)
10                 if $c_{tmp}$ is passing then
11                     $C_{pass} \leftarrow C_{pass} \cup \{c_{tmp}\}$
12                 else if $c_{tmp}$ is failing then
13                     $C_{fail} \leftarrow C_{fail} \cup \{c_{tmp}\}$
14     else
15         for $i \leftarrow 1$ to $m$ do
               /* combine the disabled options and enable
                  them */
16             $\mathbb{F} \leftarrow$CombineAndEnable($F_0, i$)
17             for $F'$ in $\mathbb{F}$ do
18                 $c_{tmp} \leftarrow$ OptionSequence($F', O_{level}$)
19                 if $c_{tmp}$ is failing then
20                     $C_{fail} \leftarrow C_{fail} \cup \{c_{tmp}\}$
21                     goto line 4
22 if Size($C_{fail}$) $=$ 0 then
23     if notEqual($F$, ReduceBugfreeOption($F$)) then
24         $F \leftarrow$ ReduceBugfreeOption($F$)
25         repeat from line 3
26     else
27         $C_{fail} \leftarrow \{O_{level}\}$
28 $C_{pass} \leftarrow C_{pass} \cup$ DisableBugrelatedOption($O_{level}$)
29 return $C_{fail}, C_{pass}$
```

3) Overall Algorithm: Algorithm 1 describes the implementation of `ODFL` and how it searches for passing and failing optimization configurations at length. In this algorithm, Line 1 initializes the search set $F = \{f_{br_1}, ..., f_{br_n}, f_{bf_1}, ..., f_{bf_m}\}$, where f_{br_j} is set to 0 and f_{bf_j} is set to 1, and n/m refers to the number of bug-related options and bug-free options respectively. F_1 and F_0 refer to the set of enabled options and the set of disabled options respectively. Line 2 constructs the optimization configuration based on the initial search set. Line 3-21 search for the failing and passing optimization configurations until achieving terminating conditions. To balance the relationship between effectiveness and efficiency, the terminating conditions are: ① no element in C_{fail} when search time exceeds 1 hour; ② more than 2 elements in C_{fail} when the search time exceeds 1.5 hours; and ③ the number of passing optimization configurations is no more than 50. We set the threshold for the number of passing configurations to 50 to save computational resources and avoid redundant passing coverage information. After acquiring the initial configuration, the algorithm enters one of the two execution units (line 4-13 or line 14-21) according to the compilation result. If the compilation result is failing, it is a failing configuration with few optimizations enabled and added into C_{fail} (line 5). Line 6-13 acquire a set of option configurations and determine whether each configuration is passing or failing. Line 7 selects i enabled options from F_1 and disables them to construct $C_{len(F_1)}^i$ optimization configurations. Line 8 traverses each optimization option configuration in $\mathbb{F}$. The passing configurations share the similar option sequence with the failing one and are added into C_{pass} (line 11). Otherwise, the new failing optimization configuration with fewer optimizations is added to C_{fail}. Lines 14-21 utilize the combination search (line 16) to enable the options in the disabled option set for the next configuration. Until the failing configuration is found, `ODFL` goes to line 4 and collects the passing and failing configurations. If there is no failing configuration constructed via line 3-21, `ODFL` reduces the bug-free options via removing the options enabled under the highest bug-free optimization level and repeats the process from line 3 (line 25). If the set of bug-free options has been reduced, the lowest bug-triggering optimization level is used as the failing optimization configuration (line 27). Line 28 adds several passing coverage to C_{pass}, which are obtained by disabling the option in bug-related set one after another. Furthermore, the coverage of the compiler under passing optimization levels is utilized to increase the diversity of coverage information.

C. Compiler Fault Localization

According to the failing and passing configurations we collected, `ODFL` locates the compiler bugs by leveraging the idea of SBFL via comparing the failing and passing coverage. Different from prior work [2], [3], we introduce multiple failing coverage in this study. Here, `ODFL` adopts the SBFL formula Ochiai [8], as shown in Formula 1, to compute the suspicious value for each touched statement:

$$value(s) = \frac{ef_s}{\sqrt{(ef_s + nf_s)(ef_s + ep_s)}} \qquad (1)$$

Where ef_s refers to the number of failing optimization configurations that execute the statement s, ep_s refers to the number of passing optimization configurations that execute the statement s, and nf_s refers to the number of failing optimization configurations that do not execute the statement s. Meanwhile, `ODFL` only considers the statements covered by the failing optimization configurations. Following the prior work [2], [3], to calculate the suspicious value of each compiler file, `ODFL` aggregates the suspicious values of all statements in a source file to represent its suspicious value:

$$VALUE(f) = \frac{\sum_{i=1}^{n_f} value(s_i)}{n_f} \qquad (2)$$

Authorized licensed use limited to: Nanjing University. Downloaded on February 23,2023 at 03:22:40 UTC from IEEE Xplore. Restrictions apply.

where n_f is the number of statements covered by failing optimization configurations in the compiler file f. ODFL generates a ranking list of compiler files based on the suspicious value of each compiler file in reverse order. The top ranked source file has a higher probability of being a buggy file.

V. Evaluation

In this study, we answer the following research questions:

- **RQ1**: How is the performance of ODFL compared with existing compiler fault localization approaches?
- **RQ2**: How is the contributions of each major component of ODFL?
- **RQ3**: How do different SBFL formulae affect the performance of ODFL?

A. Benchmark

We used the popular open-source C compiler GCC [9] as subject, which is widely used as the optimization of choice in both industry and academia [17]–[20]. To investigate the effectiveness of ODFL, 60 real GCC compilation bugs in the released benchmark are used for evaluation [3]. These 60 GCC bugs are chosen for the following reasons: First, all of them are GCC bugs in the optimization components and have been fixed by developers. Second, they are also used in prior studies as the benchmark, facilitating us to easily compare with prior studies. Meanwhile, the following information for each of the 60 GCC bugs can be obtained: the bug-triggering optimization level, the bug-triggering test program, the buggy compiler version, and the buggy compiler files. A compiler bug in the benchmark has one or more buggy files, which can be used as the ground truth to evaluate the effectiveness of our proposed approach.

B. Implementation and Configuration

GCOV. ODFL utilizes Gcov [14] to collect the coverage information of compilers. Since there are differences in coverage information obtained by different Gcov versions, we use Gcov 6.0.0 to obtain the compiler coverage information. If there is an incompatibility between GCC and Gcov 6.0.0, we select the version of Gcov that is compatible with GCC.

After confirming the version of Gcov, there is no randomness in our experiment, and thus we only need to perform the experiment once, which improves the efficiency of the experiment.

Compiler. In our study, we only conduct our experiments on GCC. The reason is that the optimization options under each optimization level in GCC are specifically accessible. At the same time, each optimization option of GCC can be manually switched, while not all passes in LLVM can be switched individually. For LLVM, which is another mature and widely-used compiler in both industry and academia, the optimization passes affect each other and the same pass are involved in different parts of the compilation process. Therefore, disabling the passes in LLVM one after another is quite different from disabling the options in GCC. Of course, there are specific solutions for LLVM pass, such as LLVM's official options for debugging and optimizing pass [21]. We will study this part in the future.

Optimization Configuration. To locate the root cause of optimization bugs, we are more concerned about the finer-grained optimization options that trigger bugs rather than bug-triggering levels. Thus, we disable the finer-grained options enabled by the bug-triggering optimization level one after another, if the compiler execution result flips (i.e., from failing to passing), we call the option a *bug-related option*; otherwise, we call it a *bug-free option*. Since the bug-related and bug-free options are enabled by default, ODFL disables the selected options according to the selection algorithm in Section IV. Then, the disabled options, along with the bug-triggering optimization level, are collectively denoted as the compiler optimization configuration. If the buggy compiler triggers the bug with a optimization configuration, the configuration is called a failing optimization configuration; otherwise, it is called a passing optimization configuration. Therefore, ODFL can obtain the failing and passing coverage information via those failing and passing optimization configurations.

Environment and Hardware. Our study was conducted in a docker container equipped with 20-core CPU, 120G memory. The operating system on the container is Ubuntu 14.04 64-bit.

C. Compared Approaches

We compare ODFL with RecBi and DiWi, where RecBi is the state-of-the-art compiler bug isolation approach. The main idea of RecBi and DiWi is that they use some strategies to generate passing test programs instead of the test programs provided by developers. The compared approaches are introduced as follows.

DiWi and RecBi. Generated passing test programs are utilized to reduce the suspicion of innocent program elements in RecBi and DiWi. These two approaches provide a new notion: instead of the passing test programs provided by developers, generating effective passing test programs is desirable. Specifically, the two main components in RecBi are structural mutation and the reinforcement learning strategy. DiWi locates compiler bugs via mutating local operates and sampling programs with Metropolis-Hasting algorithm. Moreover, compared with DiWi and RecBi, ODFL replaces the mutated programs with fine-grained optimization configurations of compilers. Thus, we also investigated whether the optimization configurations outperform the generated programs for the buggy compilers under test.

SBFL formulae. ODFL utilizes Ochiai [8] to calculate the suspicious value of statements in the compiler by default, since it has been reported to be the most effective formula for SBFL [22], [23]. Nevertheless, it is yet unknown whether ODFL can achieve good performance using other SBFL formulae since plenty of formulae based on statistical analysis have been proposed for SBFL. In this study, we compared Ochiai with other five most widely studied SBFL formulae [22]–[24] and investigated how these different formulae have impact on the performance of ODFL. Table II presents the SBFL formulae we investigated. In these formulae, ef_s,

Authorized licensed use limited to: Nanjing University. Downloaded on February 23,2023 at 03:22:40 UTC from IEEE Xplore. Restrictions apply.

nf_s, and ep_s are defined in the same way as explained in Formula 1, np_s denotes the number of passing cases that do not execute the statement s, and $totalf_s$ and $totalp_s$ denote the total number of failing cases and passing cases for statement s respectively.

TABLE II
THE SIX ADAPTED SBFL FORMULAE IN ODFL

Name	Formula
Ochiai [8]	$\frac{ef_s}{\sqrt{(ef_s+nf_s)(ef_s+ep_s)}}$
Tarantula [12]	$\frac{ef_s/totalf_s}{ef_s/totalf_s+ep_s/totalp_s}$
Ochiai2 [13]	$\frac{ef_s np_s}{\sqrt{(ef_s+ep_s)(nf_s+np_s)(ef_s+np_s)(nf_s+ep_s)}}$
Op2 [13]	$ef_s - \frac{ep_s}{(totalp_s+1)}$
Barinel [4]	$1 - \frac{ep_s}{ep_s+ef_s}$
DStar [11]	$\frac{ef_s^{\star}}{ep_s+nf_s}$

We used $\star = 2$, the most thoroughly explored value [23]

D. Measurements

A ranking list of suspicious compiler files for each compiler bug was produced by each compiler fault localization approach. We evaluate the effectiveness of compiler fault localization approaches with measuring the position of buggy files in ranking lists. Considering that multiple compiler files have the same suspicious value, we take the worst ranking as the final ranking of these files following the existing work [2], [3]. To evaluate the experimental results from different aspects, we measure the following three metrics that are widely-used by the existing work [2], [3], [25].

- **Top-N**: calculates the number of bugs successfully located within the Top-n position (i.e., n $\in \{1, 5, 10, 20\}$ in our study) of the result ranking list. Higher is better for this metric.
- **Mean First Rank (MFR)**: calculates the mean of the rank of the first buggy element (i.e., buggy file in our study) for each bug in the ranking list. The position of the first buggy element is focused on in MFR. Smaller is better for this metric.
- **Mean Average Rank (MAR)**: calculates the mean of the average rank of all buggy elements (i.e., buggy files in our study) for each bug in the ranking list. The position of all buggy elements are focused on in MAR. Smaller is better for this metric.

E. Results and Analysis

1) RQ1: ODFL v.s. Existing Compiler Fault localization Approaches: Row "ODFL" in Table III presents the effectiveness of ODFL. Overall, within Top-1, Top-5, Top-10, and Top-20 files, ODFL locates 15, 33, 48, 54 compiler bugs (out of the 60 GCC bugs) respectively. It implies that ODFL can locate nearly 25%, 55%, 80%, and 90% of the bugs effectively within Top-1, Top-5, Top-10, and Top-20 files respectively. Moreover, the MFR and MAR values of ODFL are 8.50 and 8.96 respectively. This means that, on average, the first buggy file of the buggy compiler is positioned within the first 8.50 files, and all buggy files are located within the first 8.96 files.

The comparison results among various approaches are also illustrated in Table III. From this table, we can see that ODFL indeed improves the effectiveness of compiler fault localization compared with the other approaches (RecBi and DiWi). Specifically, ODFL locates 87.5%, 57.14%, 41.18%, and 22.73% more bugs than RecBi, within Top-1, Top-5, Top-10, and Top-20 files. Also, 200%, 50%, 26.32%, and 25.58% more bugs are isolated via ODFL compared with DiWi respectively. The overall improvements of ODFL over RecBi are 45.06% and 36.55% in terms of MFR and MAR, respectively. That demonstrates that ODFL achieves much more precise localization results than RecBi and DiWi, thus reflecting the superior results achieved by ODFL compared with the state-of-the-art approaches.

2) RQ2: Contributions of the Main Components of ODFL: For further analysis, we investigated the contributions of the main components of ODFL, i.e., the failing optimization configurations and passing optimization configurations. Specifically, we designed the following three variants of ODFL to answer this question.

- **ODFL**$_{wf}$ replaces the failing optimization configurations ODFL searched with the bug-triggering optimization level to compile the test program (i.e., ODFL locates bugs **w**ithout the **f**ailing optimization configuration component. For instance, *gcc -O3* replaces *gcc -O3 -fno-f1 -fno-f2 ...*).
- **ODFL**$_{wp}$ replaces the passing coverage ODFL utilized with the passing coverage collected via generated passing test programs by RecBi (i.e., ODFL locates bugs **w**ithout the **p**assing optimization configuration component.).
- **ODFL**$_{wfp}$ replaces both the failing and passing optimization configurations with the bug-triggering and bug-free optimization levels to compile test programs (i.e., ODFL locates bugs **w**ithout both the **f**ailing and **p**assing optimization configuration components)

We compared ODFL with ODFL$_{wf}$ to investigate the contribution of our designed failing optimization configurations. Besides, we compared ODFL with ODFL$_{wp}$ to investigate the contribution of our designed passing optimization configurations. Moreover, we compare ODFL with ODFL$_{wfp}$ to investigate the effectiveness of our configuration strategy applied for compiler fault localization. In the following, we introduce the contributions of the main components of ODFL at length.

Contribution of Failing Optimization Configurations. The "ODFL$_{wf}$" row in Table III presents the localization result of ODFL without the component of failing optimization configurations. Bug-triggering optimization levels enable all sub-optimization options by default, while the generated failing optimization configurations use fewer optimizations via disabling some innocent options. From Table III, the Top-1 metric is significantly improved (from 1 to 15) with the support of the failing optimization configuration component via comparing ODFL and ODFL$_{wf}$. Moreover, the improvements for Top-5, Top-10, Top-20, MFR, and MAR are 57.14%, 71.43%, 45.95%, 54.25%, and 54.42% respectively. Such results reflect the immense localization capability of the failing optimization configurations generated by ODFL. Specifically, ODFL tries

Authorized licensed use limited to: Nanjing University. Downloaded on February 23,2023 at 03:22:40 UTC from IEEE Xplore. Restrictions apply.

TABLE III
COMPILER FAULT LOCATION EFFECTIVENESS COMPARISON

Approach	Top-1	$\Uparrow_{Top-1}$	Top-5	$\Uparrow_{Top-5}$	Top-10	$\Uparrow_{Top-10}$	Top-20	$\Uparrow_{Top-20}$	MFR	$\Uparrow_{MFR}$	MAR	$\Uparrow_{MAR}$
ODFL	**15**	—	**33**	—	**48**	—	**54**	—	**8.50**	—	**8.96**	—
RecBi	8	87.50	21	57.14	34	41.18	44	22.73	15.47	45.06	14.12	36.55
DiWi	5	200.0	22	50.00	38	26.32	43	25.58	16.25	47.69	16.81	46.70
ODFL$_{wf}$	4	275.0	21	57.14	28	71.43	37	45.95	18.58	54.25	19.66	54.42
ODFL$_{wp}$	2	650.0	17	94.12	29	65.52	37	45.95	23.02	63.08	23.14	61.28
ODFL$_{wfp}$	0	∞	3	1000	20	140.0	32	68.75	27.32	68.89	27.89	67.87

"$\Uparrow_*$" refers to the improvement rate of ODFL over a compared approach in terms of the metric "$*$".

its best to find the failing configurations that enable few optimizations, and eliminates the suspiciousness of innocent optimizations to the greatest extent. In this way, ODFL is able to achieve more efficient and accurate fault localization capabilities.

Contribution of Passing Optimization Configurations. The "ODFL$_{wp}$" row in Table III presents the localization result of ODFL without the component of passing optimization configurations. Specifically, ODFL$_{wp}$ replaces the passing configurations with the passing programs generated in RecBi, to evaluate the effectiveness of the passing optimization configurations used in ODFL. RecBi repeatedly generated passing test programs for 5 times in its experimental setup and calculated the median results to reduce the influence of randomness. On the contrary, the same passing optimization configurations are obtained in ODFL under the same experimental setup, which eliminates the influence of randomness. In ODFL$_{wp}$, to make full use of the programs generated by RecBi, we use all programs generated in 5 rounds of the experiments of RecBi to replace our passing optimization configurations. From Table III, ODFL$_{wp}$ only locates 2, 17, 29, and 37 compiler bugs within Top-1, Top-5, Top-10, and Top-20 files. Corresponding, ODFL locates 650%, 94.12%, 65.52%, and 45.95% more bugs than ODFL$_{wp}$, within Top-1, Top-5, Top-10, and Top-20 files. It indicates that the passing optimization configurations generated by ODFL not only eliminate randomness, but are also much more effective than the program generated by mutation.

Contribution of Configuring-based Compiler Fault localization. The "ODFL$_{wfp}$" row in Table III presents the localization results of ODFL without the components of both failing and passing optimization configurations. Specifically, ODFL$_{wfp}$ replaces the generated failing configurations with bug-triggering optimization levels, and replaces the generated passing configurations with bug-free optimization levels, to evaluate the overall effectiveness of our elaborate optimization configurations. As shown in Table III, ODFL$_{wfp}$ cannot locate any bugs in Top-1 compiler file. ODFL$_{wfp}$ also underperformed on several other metrics. That means looking for some optimization configurations randomly is invalid. By contrast, in ODFL, the deliberately designed optimization configuration selection for failing and passing coverage of compiler plays a great positive role.

3) RQ3: Effect of Different Formulae in ODFL: We further studied how different formulae affect the performance of ODFL and evaluated them with Top-1, Top-5, Top-10, Top-20, MAR and MFR. Table IV presents the results of ODFL using different formulae. In this table, different rows represent the different SBFL formulae utilized and different columns represent the different metrics evaluated. In terms of Top-N, ODFL achieves better performance than BecRi and DiWi with all six SBFL formulae. It demonstrates that our optimization configurations designed in ODFL outperforms than the mutated programs in RecBi and DiWi, regardless of the SBFL formula used. Meanwhile, ODFL achieves almost similar results when adopting the formulae Ochiai, Tarantula, Ochiai2. This also explains most of the formulas have no obvious effect on the performance of ODFL and reveals the effectiveness of ODFL. Moreover, adopting DStar in ODFL achieves the optimum performance in terms of Top-1, Top-5, Top-20, as shown in Table IV. However, all these formulae are not specifically designed for locating compiler bugs, and whether there are more effective formulae can be utilized remains to be unknown. We leave such exploration as our future work.

TABLE IV
PERFORMANCE OF ODFL USING DIFFERENT FORMULAE

Formula	Top-1	Top-5	Top-10	Top-20	MAR	MFR
Ochiai	15	33	**48**	**54**	8.50	8.96
Tarantula	13	35	41	51	9.82	10.24
Ochiai2	15	33	**48**	**54**	8.50	8.95
Op2	12	30	38	42	41.25	41.67
Barinel	16	36	**48**	**54**	**7.90**	**8.38**
DStar	**22**	**40**	47	**54**	16.83	17.37

TABLE V
TIME CONSUMED COMPARISON (HOURS)

Approach	Component	Avg	Max	Min
ODFL	Initializing	0.0127	0.0407	0.0039
	Searching	0.1808	1.5000	0.0009
	All	**0.5841**	**1.5000**	**0.0281**
RecBi	All	5	5	5

VI. DISCUSSION

A. *Efficiency of ODFL*

We discuss the efficiency of our ODFL by comparing the time consumed in compiler fault localization with RecBi (DiWi consumes the same time as RecBi). Specifically, we evaluated the time required for each component and entire process (including initialing the option set, searching configurations and collecting the compiler coverage) in ODFL. To

Authorized licensed use limited to: Nanjing University. Downloaded on February 23,2023 at 03:22:40 UTC from IEEE Xplore. Restrictions apply.

fully reproduce RecBi, we repeated RecBi 5 times and used the median results to reduce the influence of randomness. The termination condition of each experiment for RecBi was a 1-hour limit. The statistics of the results are summarized in Table V. As it demonstrates, ODFL can locate bugs in a buggy compiler within 0.5841 hours on average. In contrast, RecBi needs 5 hours to conduct a compiler fault localization since it repeated 5 rounds of experiments. Thus, compared with the state-of-the-art compiler fault localization approach RecBi, the efficiency of ODFL outperforms it nearly 88.32%.

B. Threats to Validity

First, we used the tools released by the existing work to run RecBi and DiWi, which reduces the threat of the effectiveness of our empirical studies. Second, although our algorithm efficiently collects failing configurations that enable as few optimizations as possible in most cases, there are still rare cases where it takes a long time to find the failing configuration due to the high complexity of the combinatorial search. To mitigate this threat, we set certain conditions to terminate the algorithm (more details explained in IV-B3). In addition, to ensure completeness of the algorithm, the lowest bug-triggering optimization level is used as the failing optimization configuration when there is no failing configuration searched. Third, to evaluate bugs in compiler optimizations and compare with existing methods, we chose the benchmark released in RecBi and all the bugs in the benchmark are optimization bugs. Specifically, there are 60 real-world optimization bugs of GCC, which ensures the credibility of our evaluation. Another threat is that our approach was only performed on GCC, and we will apply ODFL on LLVM in the future.

C. Future Works

In the future, the following aspects are worthy of further consideration. First, as the results are shown in Table IV, we observed that adopting DStar [11] in ODFL achieves the best performance on Top-N metrics. Meanwhile, we also replaced the Ochiai [8] in RecBi and DiWi with DStar and investigated whether DStar can have a positive impact on the performance of these two approaches. The experimental results demonstrated that DStar can improve both the effectiveness of fault localization of compilers in RecBi and DiWi. These indicate that it is recommended to apply DStar to locate compiler bugs. We have made it our future work to design a specific SBFL formula for ODFL. Second, since ODFL performs excellent on file-level fault localization of compilers, we will apply ODFL on finer-grained levels to locate bugs of compilers by introducing something novel and elaborate.

VII. Related Work

A. Automated Fault Localization

Fault localization research generates a large body of literature on this topic, such as spectrum-based [4], [5], [24], slicing-based [26], [27], mutation-based [28]–[30], and model-based fault localization [31], [32]. Unfortunately, due to the complexity of compilers, the above approaches can hardly be applied to this field. Recently, Chen et al. proposed DiWi [2] and RecBi [3] to facilitate compiler bug isolation, which introduce the concept of the witness program. However, they are still compromised by the limitations of effectiveness and efficiency. ODFL is proposed as a novel fault localization approach special to compilers to enhance existing techniques. Experiments show that our approach ODFL significantly outperforms RecBi and DiWi, and all SBFL formulae achieve similar results, which reflects the effectiveness of our proposed optimization configuration.

Besides, there are some researches related to automated compiler debugging. For instance, some work proposed to reduce the failing test programs to facilitate debugging [33]–[36]. Holmes and Groce proposed new metrics to measure the distance between failing test programs for facilitating compiler debugging [37], [38]. In contrast, our work focuses on configuring fine-grained optimization options to effectively locate compiler bugs.

B. Compiler Optimization

In recent years, compiler optimization settings have become a widely-used tool for diverse researches. Ren et al. studied the effectiveness of compiler optimization on binary code differences [39]. Chen et al. proposed an efficient compiler auto-tuning method based on Bayesian optimization [40]. Moreover, several empirical studies on compiler bugs were presented [1], [41]. Zhou et al. noted that optimization is the most buggy component of compilers [1]. Therefore, it is meaningful to locate optimization bugs of compilers effectively. Compared with the time-consuming approaches such as RecBi [3], we view ODFL as an effective and efficient approach to locate optimization bugs in compilers via differentiating optimization options.

VIII. Conclusion

In this study, we propose ODFL, a novel fault localization technique for locating optimization bugs of compilers via differentiating fine-grained optimization options. ODFL introduces a novel concept, configuring optimization options, to obtain several failing and passing coverage of compilers, which is based on the bug-free and bug-related options it collects. Our results demonstrate the ODFL significantly outperforms all the compared approaches in terms of all metrics. Moreover, ODFL is much more efficient than the state-of-the-art approach as it can save more than 88% time for locating bugs on average.

Acknowledgment

We sincerely thank all the anonymous reviewers for their constructive comments. We are also very grateful to J. Chen, J. Han, P. Sun, L. Zhang, D. Hao, and L. Zhang for sharing their experimental artifacts. This work was supported by the National Natural Science Foundation of China (62072194, 61772259, 62002125). We would also like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. Yibiao Yang and Ming Wen are the corresponding authors.

Authorized licensed use limited to: Nanjing University. Downloaded on February 23,2023 at 03:22:40 UTC from IEEE Xplore. Restrictions apply.

References

[1] Z. Zhou, Z. Ren, G. Gao, and H. Jiang, "An empirical study of optimization bugs in GCC and LLVM," *J. Syst. Softw.*, vol. 174, p. 110884, 2021.

[2] J. Chen, J. Han, P. Sun, L. Zhang, D. Hao, and L. Zhang, "Compiler bug isolation via effective witness test program generation," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, 2019, pp. 223–234.

[3] J. Chen, H. Ma, and L. Zhang, "Enhanced compiler bug isolation via memoized search," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, 2020, pp. 78–89.

[4] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "Spectrum-based multiple fault localization," in *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, 2009, pp. 88–99.

[5] C. M. Tang, W. K. Chan, Y. Yu, and Z. Zhang, "Accuracy graphs of spectrum-based fault localization formulas," *IEEE Trans. Reliab.*, vol. 66, no. 2, pp. 403–424, 2017.

[6] R. E. Kass, B. P. Carlin, A. Gelman, and R. M. Neal, "Markov chain monte carlo in practice: a roundtable discussion," *The American Statistician*, vol. 52, no. 2, pp. 93–100, 1998.

[7] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *J. Artif. Intell. Res.*, vol. 4, pp. 237–285, 1996.

[8] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, 2007, pp. 89–98.

[9] "Gcc," https://gcc.gnu.org/, Accessed: 2021.

[10] "Llvm," https://llvm.org/, Accessed: 2021.

[11] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Trans. Reliab.*, vol. 63, no. 1, pp. 290–308, 2014.

[12] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, 2005, pp. 273–282.

[13] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 11:1–11:32, 2011.

[14] "Gcov," https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, Accessed: 2021.

[15] F. Keller, L. Grunske, S. Heiden, A. Filieri, A. van Hoorn, and D. Lo, "A critical evaluation of spectrum-based fault localization techniques on a large-scale software system," in *2017 IEEE International Conference on Software Quality, Reliability and Security, QRS 2017, Prague, Czech Republic, July 25-29, 2017*, 2017, pp. 114–125.

[16] "Gcc bug 58570," https://llvm.org/docs/OptBisect.html, Accessed: 2021.

[17] T. Glek and J. Hubicka, "Optimizing real world applications with GCC link time optimization," *CoRR*, vol. abs/1010.2196, 2010.

[18] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather *et al.*, "Milepost gcc: machine learning based research compiler," in *GCC summit*, 2008.

[19] L. P. Cáceres, F. Pagnozzi, A. Franzin, and T. Stützle, "Automatic configuration of GCC using irace," in *Artificial Evolution - 13th International Conference, Évolution Artificielle, EA 2017, Paris, France, October 25-27, 2017, Revised Selected Papers*, 2017, pp. 202–216.

[20] D. Berlin, D. Edelsohn, and S. Pop, "High-level loop optimizations for gcc," in *Proceedings of the 2004 GCC Developers Summit*, 2004, pp. 37–54.

[21] "Optbisect," https://gcc.gnu.org/bugzilla/show_bug.cgi?id=58570, Accessed: 2021.

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

[23] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 609–620.

[24] M. Wen, J. Chen, Y. Tian, R. Wu, D. Hao, S. Han, and S. Cheung, "Historical spectrum based fault localization," *IEEE Trans. Software Eng.*, vol. 47, no. 11, pp. 2348–2368, 2021.

[25] J. Sohn and S. Yoo, "FLUCCS: using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, 2017, pp. 273–283.

[26] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Debugging with dynamic slicing and backtracking," *Softw. Pract. Exp.*, vol. 23, no. 6, pp. 589–616, 1993.

[27] X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental evaluation of using dynamic slices for fault location," in *Proceedings of the Sixth International Workshop on Automated Debugging, AADEBUG 2005, Monterey, California, USA, September 19-21, 2005*, 2005, pp. 33–42.

[28] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, 2014, pp. 153–162.

[29] S. Hong, B. Lee, T. Kwak, Y. Jeon, B. Ko, Y. Kim, and M. Kim, "Mutation-based fault localization for real-world multilingual programs (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, 2015, pp. 464–475.

[30] M. Papadakis and Y. L. Traon, "Metallaxis-fl: mutation-based fault localization," *Softw. Test. Verification Reliab.*, vol. 25, no. 5-7, pp. 605–628, 2015.

[31] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "An observation-based model for fault localization," in *Proceedings of the 2008 International Workshop on Dynamic Analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), WODA 2008, Seattle, Washington, USA, July 21, 2008*, 2008, pp. 64–70.

[32] M. Wen, R. Wu, and S. Cheung, "Locus: locating bugs from software changes," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, 2016, pp. 262–273.

[33] J. M. Caron and P. A. Darnell, "Bugfind: A tool for debugging optimizing compilers," *ACM SIGPLAN Notices*, vol. 25, no. 1, pp. 17–22, 1990.

[34] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 183–200, 2002.

[35] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, 2012, pp. 335–346.

[36] S. Herfert, J. Patra, and M. Pradel, "Automatically reducing tree-structured test inputs," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, 2017, pp. 861–871.

[37] J. Holmes and A. Groce, "Causal distance-metric-based assistance for debugging after compiler fuzzing," in *29th IEEE International Symposium on Software Reliability Engineering, ISSRE 2018, Memphis, TN, USA, October 15-18, 2018*, 2018, pp. 166–177.

[38] J. Holmes and A. Groce, "Using mutants to help developers distinguish and debug (compiler) faults," *Softw. Test. Verification Reliab.*, vol. 30, no. 2, 2020.

[39] X. Ren, M. Ho, J. Ming, Y. Lei, and L. Li, "Unleashing the hidden power of compiler optimization on binary code difference: an empirical study," in *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, 2021, pp. 142–157.

[40] J. Chen, N. Xu, P. Chen, and H. Zhang, "Efficient compiler autotuning via bayesian optimization," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, 2021, pp. 1198–1209.

[41] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in GCC and LLVM," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. ACM, 2016, pp. 294–305.

Authorized licensed use limited to: Nanjing University. Downloaded on February 23,2023 at 03:22:40 UTC from IEEE Xplore. Restrictions apply.