JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 9, No. 3, September 2015, pp. 142-154

Selecting Test Cases for Result Inspection to Support Effective Fault Localization

Yihan Li[1,2]*, Jicheng Chen[1], Fan Ni[1,2], Yaqian Zhao[1,2], and Hongwei Wang[1]

[1]State Key Laboratory of High-end Server & Storage Technology, Inspur, Beijing, China, and
[2]School of Computer Science and Engineering, Beihang University, Beijing, China

liyihannew@126.com, chenjch@inspur.com, nifan@inspur.com, zhaoyaqian@inspur.com, wanghongwei@inspur.com

Abstract

Fault localization techniques help locate faults in source codes by exploiting collected test information and have shown promising results. To precisely locate faults, the techniques require a large number of test cases that sufficiently exercise the executable statements together with the label information of each test case as a failure or a success. However, during the process of software development, developers may not have high-coverage test cases to effectively locate faults. With the test case generation techniques, a large number of test cases without expected outputs can be automatically generated. Whereas the execution results for generated test cases need to be inspected by developers, which brings much manual effort and potentially hampers fault-localization effectiveness. To address this problem, this paper presents a method to select a few test cases from a number of test cases without expected outputs for result inspection, and in the meantime selected test cases can still support effective fault localization. The experimental results show that our approach can significantly reduce the number of test cases that need to be inspected by developers and the effectiveness of fault localization techniques is close to that of whole test cases.

Category: Smart and intelligent computing

Keywords: Fault localization; Debugging; Test case selection; Passed tests; Failed tests

I. INTRODUCTION

Software testing plays an indispensable role in guaranteeing the quality of software in software engineering. Once the program fails during the process of testing, developers often use debugging methods to find and eliminate faults in the program. Traditionally, developers often manually insert output statements in the program or set breakpoints using debugging tools to find the location of faults. Both the methods require developers to have a deep understanding of the structure of the program. Further research [1] has indicated that traditional debugging is a very tedious and time-consuming activity in software testing, which becomes a heavy burden for developers. In order to reduce the heavy labor of manual debugging, many coverage-based fault localization (CBFL) techniques [2-4] have been proposed. Typically, such techniques first collect dynamic execution information when the program is exercised by test cases. Then based on the collected test information, including coverage information and execution results, they calculate the likelihood of program statements that contain faults according to statistical models. The basic intuition of statistical models is that faulty statements are correlated higher with program

Open Access http://dx.doi.org/10.5626/JCSE.2015.9.3.142 http://jcse.kiise.org

Received 30 August 2015; **Accepted** 07 September 2015
*Corresponding Author

 pISSN: 1976-4677 eISSN: 2093-8020

failure via coverage information and hence statements that are executed by more failed test cases and fewer passed test cases are often more suspicious. Finally, program statements are ranked in descending order according to the suspicion of containing faults, and statements that are ranked higher are given a higher priority for developers to check whether the statements contain fault or not. Recent researches [5-8] show that such fault localization techniques can effectively reduce the number of statements that need to be checked, thereby reducing the manual labor of debugging.

To effectively locate faults in the program, CBFL techniques require a large number of test cases with high coverage of program statements together with execution results of each test case. However, in some debugging scenarios, such as processes of software development, there are only a few test cases with execution results available for CBFL techniques. If only a few test cases are used for CBFL, it is not very effective [3]. As the development of test case generation techniques [9, 10], a large number of test cases with high coverage of statements can be automatically generated. However, the execution results of test cases still need to be labeled as a failure or a success. Determining whether the program behaves correctly for a test case is the problem of test oracles [11]. Traditionally developers can manually inspect execution results for all test cases. However, it is very time-consuming and tedious to finish such work and it increases the burden for applying CBFL techniques in practice. To automatically determine the test oracles, researchers [12] have proposed to construct automated test oracles. However, for most programs, constructing automated test oracles is very complicated and still requires much manual effort. As a result, in order to apply CBFL techniques effectively when execution results of many test cases are not available, the test oracles will cause additional manual effort, which would severely increase the cost of debugging.

To alleviate the burden of applying CBFL in such scenarios, this paper will study how to select a small number of test cases for further result inspection from a large number of test cases without test oracles. It is expected that the selected test cases are a small portion of the original test suite but still can provide sufficient information for locating faults. The approach proposed in this paper uses a dynamic basic block as units to represent each test case. Once there exists any test case that can divide the set of dynamic basic blocks, such test cases are candidates for result inspection. The approach tends to select the test case that can divide the set of dynamic basic blocks evenly as well as cover more dynamic basic blocks that are exercised by a high rate of failed test cases. Experimental results show that our approach can select only a few test cases for inspection while the effectiveness of fault localization is close to that when whole test cases are used.

The rest of this paper is organized as follows: Section II describes related work to fault localization. Section III presents the current approach in detail. Section IV describes experimental design. Section V presents the empirical results and analysis. Section VI discusses some issues in the approach. Finally, a conclusion to the results is presented in Section VII.

II. RELATED WORK

In this section, we briefly review previous studies related to fault localization.

Since this paper focuses on selecting a small portion of original test suites, our work is somewhat related to test selection and prioritization techniques. Researchers have conducted extensive work to study test selection and prioritization techniques. Test selection aims at selecting some tests from a given test suite for retesting a modified program by focusing on the modified code, whereas test prioritization schedules tests for execution in order to achieve some specific goal. Since existing test selection techniques aim at improving efficiency of software testing, they may not be effective at improving fault-localization effectiveness [8]. Meanwhile, some researchers try to improve test prioritization techniques so that prioritized tests can be used for effective fault localization. Gonzalez-Sanchez et al. [13] proposed a test prioritization based on information gain. Their experimental results indicate fault-localization effectiveness based on 47% of the top test cases that are close to that of the entire set of test cases. However, the method depends on the estimation of distribution of faults, which is difficult to obtain. To solve the above problem, Gonzalez-Sanchez et al. [14] further proposed a test prioritization technique based on an ambiguity group. Their technique groups together the statements with same coverage information, and then construct a probability model on the likelihood of each group containing faulty. Experimental results indicate that in most cases the method is more effective. Gong et al. [15] proposed a test case prioritization strategy called Diversity Maximization Speedup. The strategy orders a set of unlabeled test cases in a way that maximizes the effectiveness of a fault localization technique. However, the strategy requires some test cases to be labeled by developers in the initial stage, which may limit its scope of application. Although test case prioritization techniques toward fault localization can prioritize test cases in a way that a small portion of test cases can achieve considerable fault localization effectiveness, it still requires developers to determine how many test cases should be used for fault localization.

The test oracle problem often hampers the fault-localization effectiveness when there are no sufficient test cases with test oracles available for fault localization. Artzi et al. [16] proposed a new directed test-generation

technique to generate small test suites for fault localization. They showed that the technique preserves fault-localization effectiveness while reducing the test-suite size by 86.1%. To mitigate the impact of the test oracle problem on fault localization, Xie et al. [12] applied a metamorphic slice to CBFL techniques. However, metamorphic testing requires developers to have a deep understanding of programs, and thus it may not be effective for complex programs. Hao et al. [17] proposed three strategies on reducing test cases for result inspection to save the effort of developers from tedious test-result checking when applying CBFL techniques. The basic idea is that statements can be divided into different equivalence classes based on coverage information. In their strategies, test cases that satisfy one of the following conditions are reserved: 1) the test case that can generate more equivalence classes; 2) the test case that can evenly generate equivalence classes; 3) the test case that can generate more equivalence classes which are covered by more failed test cases. Once there are no test cases that can divide equivalence classes, the test selection process terminates. Their experimental results indicate that their approach can help developers inspect the result of a smaller subset of test inputs, whose fault-localization effectiveness is close to that of the whole test collection. However, equivalence classes are based on statements, which may not distinguish statements properly. Hence the strategies may still yield too many test cases for result inspection.

In this paper, we extend our previous work [18] to present more details about our approach and conduct additional experiments on more subject programs, including large-scale programs, which are conducted to verify the effectiveness of the approach. Further discussions on our approach are also presented.

III. OVERVIEW OF PROPOSED APPROACH

Since debugging activity often starts when developers have at least one failed test case beforehand, it is assumed that developers have initially had one failed test case, which will facilitate the subsequent discussion. Typically, there may exist some test cases in which developers have checked test results before encountering at least one failed test case. The common debugging scenario would include some failed test cases along with several passed test cases. However, such situations where there are several test cases and at least one failed test case can be viewed as a situation occurring during our test selection process. Thus, when presenting our approach, it is assumed that there is only one failed test case in the beginning of our approach, which is denoted as t_s.

Fig. 1 depicts an overview of our approach applied to CBFL. The approach consists of three steps. 1) Preprocess: a large number of test cases without test oracle are generated by a test case generation tool, and the coverage information is collected automatically in this step. 2) Test

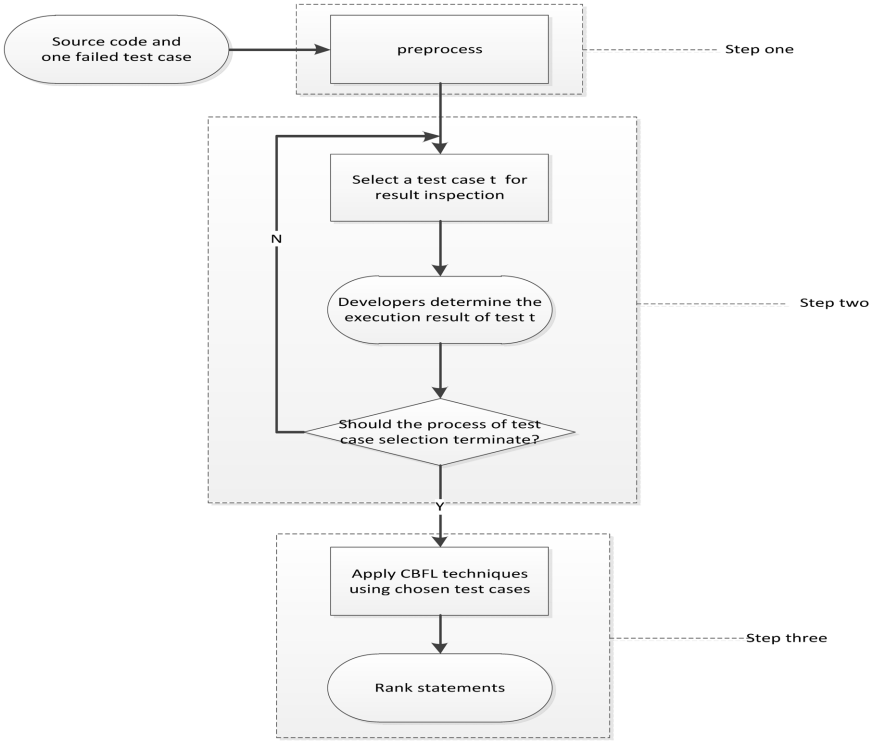

Fig. 1. Overview of the proposed method applied to CBFL.

case selection: a test case is selected for result inspection at a specific time. Developers label the selected test case as passed if its result is the same as the expected output test or failed if it is not. Once the selected test cases are sufficient for fault localization, the selection process terminates. Otherwise the selection process continues. 3) Fault localization: the selected test cases, along with the initial failed test case, are used for fault localization.

The following subsection will describe our approach in detail.

A. Preprocess

The basic idea behind CBFL techniques is that they exploit how program entities are correlated with program failures via statistically analyzing coverage information. If the test cases are lacking, it may fail to distinguish between faulty statements and non-faulty statements by coverage information. Thus, it is expected that selected test cases can provide sufficient coverage information in order to distinguish statements. According to such principle, in the beginning of our approach a large number of test cases are generated by a test case generation tool to obtain sufficient coverage information, which are then used as a test suite for a later selection procedure. For the subsequent discussion, the generated test cases are denoted as T_g; however test results are not available at this step.

Suppose the faulty program P consists of m statements, which can be denoted as $P = \{s_1, s_2, s_3, ..., s_m\}$. The test set running against P is denoted as $T = T_g \cup t_s$. The coverage information is collected through the stub method automatically after execution of every test case in T. In this paper, statement coverage is collected and the information is represented as a $n \times m$ matrix MS, in which n is the size of the test set T and m is the number of statements in program P. Each element c_{ij} in MS denotes whether the statements s_j are covered by the test t_i or not, and whose value is set to 1 if it is covered, otherwise it is 0.

Thus, a large number of test cases with coverage information are obtained at this step, but the test results are not available. For the subsequent discussion, we denote the test set in which test cases have been selected and test results have been inspected as T_r. Especially, in the beginning of the approach, $T_r = \{t_s\}$.

B. Test Case Selection Strategy

1) Dynamic Basic Block

Our approach aims at selecting as few test cases as possible to distinguish faulty statements from non-faulty statements. Essentially, different levels of coverage make statements distinguishable from each other. However, some of the statements in the program could always be exercised by the same set of test cases in the test suite T. The CBFL techniques may fail to distinguish such statements. Note that it is important to group such statements together, and thus based on such sets of statements it will be clearer to measure the ability of test cases to distinguish the statements.

In this paper, we will adopt the concept of dynamic basic block (DBB), proposed by Baudry et al. [19], to represent such sets of statements in the program.

DEFINITION 1 (Dynamic basic block). *A dynamic basic block (DBB) is a set of statements in the program P that are all covered by the same test cases in the test suite T.*

According to the definition of dynamic basic block, it can be easily constructed once the coverage information of the test suite T is available. After the set of dynamic basic blocks is constructed based on the coverage information, our approach will represent each test case based on dynamic basic blocks. Specifically, the following procedure is conducted.

(1) A set of dynamic basic blocks is generated according to the coverage information of test cases in T, which is denoted as $B(T)=\{B_1, B_2, B_3, ..., B_w\}$ and B_i represents a dynamic basic block.

(2) Each test case t_i in T is represented as a W-dimensional vector according to whether t_i exercises the corresponding dynamic basic block or not during its execution: where b_{ij} ($1 \le j \le w$) is set to 1 if t_i covers the dynamic basic block B_j otherwise its value is set to 0.

$$t_i = < b_{i1}, b_{i2}, ..., b_{iw} >$$

2) Group and Indistinguishable Group

Note that the statements in the same dynamic basic block are exercised by the same test cases; thus after representing test cases using $B(T)$, the ability of test cases to distinguish statements will be based on dynamic basic blocks.

In general, to distinguish two dynamic basic blocks, it is expected that these two dynamic basic blocks are executed either by a different number of failed test cases or by a different number of passed test cases; otherwise it is difficult for CBFL techniques to distinguish these two dynamic basic blocks. To further discuss our approach, some concepts are presented to capture such features.

DEFINITION 2 (Group). *A set of dynamic basic blocks g is called a group if and only if it satisfies the following condition: any two dynamic basic blocks in the set g are covered by the same number of failed test cases and the same number of passed test cases in the test set T_r*

DEFINITION 3 (Indistinguishable group). *A group g is called an indistinguishable group if and only if it satisfies the following condition: if u is an element of the group g, and v is not an elements of the group g, then there exists*

at least one test case in the test set T_r that it covers either u or v.

According to the above two definitions, the set of groups $G(T_r)$ generated by test set T_r is not unique, while the set of indistinguishable groups $IG(T_r)$ generated by T_r is uniquely determined. To uniquely determine the set of groups for the test set T_r, we set $G(T_r)$ equal to $IG(T_r)$ before the test selection process starts. Given a test case t in T_g, for a group g in the current $G(T_r)$, if t partially covers some dynamic basic blocks in g, we refer to this as that the test t divides the group g. In this situation, g can be split into two sub-groups based on coverage information, one of which is covered by t while the other is not. Note that only such test cases in T_g are useful for fault localization, since they divide some groups in $G(T_r)$ and thus more dynamic basic blocks may be distinguished from each other.

3) Detail Approach

We first present how our proposed approach selects a test case from T_g at a time for result inspection, which is based on two basic intuitions. An effective selection strategy should not only reduce the size of selected test cases, but also support fault localization when the selected test cases are used.

To reduce the size of the selected test cases, it is expected that after selecting a few test cases from T_g, the current groups in $G(T_r)$ can't be divided by any test cases in T_g. Consider a test case t in T_g; suppose t can divide some groups in $G(T_r)$, which are denoted as $DG(T_r, t)$. For any group g_i in $DG(T_r, t)$, g_i is divided into two sub-groups g_{i1} and g_{i2}. Denote $|g_i|$ as the number of dynamic basic blocks in group g_i. If the difference between $|g_{i1}|$ and $|g_{i2}|$ is large, it may have to select more test cases from T_g to divide the larger sub-group. If the difference is small, fewer test cases may be needed to divide the sub-groups. Thus, for any group g_i in $DG(T_r, t)$, it is expected that the selected test case t can divide g_i evenly such that fewer test cases are selected out for the result inspection. The following formula is first introduced:

$$Split(t, g_i) = \min(|g_{i1}|, |g_{i2}|) \quad (1)$$

$Split(t, g_i)$ represents the number of dynamic basic blocks in the smaller sub-group. If t does not divide the group g_i, then $Split(t, g_i)$ equals to 0. This formula will be then used in a later strategy to assess evenness of division.

To support fault localization, it is expected that selected test cases can distinguish non-faulty statements from faulty statements as much as possible. Specifically, if the selected test cases can divide groups that potentially contain faulty statements, then such division is more useful for fault localization. Thus, we use a method to calculate how likely each group in $G(T_r)$ contains faulty statements. In this paper, O^p metric [4] is used. The formula of O^p is as follows:

$$S(g_i) = a_{ef}(g_i) - \frac{a_{ep}(g_i)}{PT+1} \quad (2)$$

Where $a_{ef}(g_i)$ is the number of failed test cases in T_r that cover the group g_i and $a_{ep}(g_i)$ is the number of passed test cases in T_r that cover the group g. PT is the number of passed test cases in T_r. $S(g_i)$ calculates how likely the group g_i are correlated with faults based on test information of T_r.

Thus, our selection strategy is based on above two intuitions. Suppose the groups of the current $G(T_r)$ can be denoted as $\{g_1, g_2, ..., g_z\}$. When a test case t in T_g is chosen for result inspection, the test case t must satisfy the following formula:

$$\arg\max_{t \in T_g} \left\{ \sum_{i=1}^{|G(T_g)|} S(g_i) * Split(t, g_i) \right\} \quad (3)$$

As can be seen from the formula, $Split(t, g_i)$ guides the strategy to select the test case that can divide groups evenly while $S(g_i)$ guides the strategy to select the test case that can divide groups which potentially contain faulty statements.

Based on the above analysis, Fig. 2 presents a detailed selection algorithm for our approach. The algorithm requires one failed test case as well as a large number of test cases without test oracle as inputs. The codes in lines 1-3 add the initial failed test case to T_r, and then generate

Algorithm 1 Test Selection Procedure

Input:

t_s – an initial failed test case

T_g – test set in which test results are unknown

MS – statement coverage information for $T = t_s \cup T_g$.

Output:

T_r – the set of selected test cases

Procedure :

```
1 :  Tr ←{ts}
2:   Generate the set of dynamic basic blocks B(T) according to MS
3:   ∀t∈ T, represent t as vector using units of B(T)
4:   while true
5:       G(Tr) ← IG(Tr)
6:       while true
7:           ∀gi∈G(Tr) , calculate s(gi) according to formula (2)
8:           a test case t is selected from Tg according to formula (3)
9:           if (|G(Tr∪t)| == |G(Tr)|)
10:              break;
11 :         developers manually label the test case t as failed test or passed test.
12:          Tr = Tr∪ t
13:          Tg = Tg – {t}
14:      end while
15:      If (|G(Tr)| == |IG(Tr)|)
16:          break;
17: end while
18: return Tr
```

Fig. 2. Test selection algorithm to support fault localization.

$B(T)$ based on coverage information of T. In line 5 the group $G(T_r)$ is initialized to $IG(T_r)$. Then in lines 6-14 test cases in T_g are selected iteratively according to the formula (2). Lines 9-10 check whether the selected test case can divide some groups in $G(T_r)$ or not. If division occurs, lines 12-13 add the selected test case to T_r and remove it from T_g. If no division occurs, the selection procedure breaks. Line 16 further checks whether the size of indistinguishable groups generated by the current T_r is equal to that of $G(T_r)$. If it is equal, it indicates that the groups in $IG(T_r)$ cannot be divided by T_g any more. In such situations the algorithm returns the selected test cases. If it does not, it indicates that there may still exist some test cases in T_g that can divide indistinguishable groups in $IG(T_r)$, then the algorithm returns to line 5 and continues to select test cases.

C. Fault Localization

In this paper, two representative fault localization techniques are used to investigate the impact of the proposed approach. *Tarantula* is proposed by Jones et al. [2]; the technique gives high suspicion to the statements that are covered by many failed tests and few passed tests. Different from *Tarantula*, *Ochiai* [3] also takes the absence in failed executions into account and gives high weight to failed tests in calculating suspicion. Since the focus of this paper is not on fault localization techniques, only two representative techniques are used. In future works, we plan to investigate the impact of the proposed approach on more fault localization techniques.

IV. EXPERIMENTAL DESIGN

A. Subject Programs

To evaluate whether our approach can work well, *Siemens Test Suite* and *Unix* programs are both used as our subject programs. *Siemens Test Suite* contains seven small programs and *Unix* contains three large programs. These programs have been extensively used to evaluate effectiveness of CBFL techniques in previous studies [2-5]. Table 1 lists the detailed information for each program. The *Versions* column lists the number of faulty versions for each subject program. The column *LOC* shows the lines of code for each program. The column *Size of TS* represents the total number of available test cases in the test pool for each program.

To validate the effectiveness of the proposed method, first a failed test case is chosen randomly from the test suite T from each faulty version of the program, which is used as the initial failed test case t_s, and then the remaining test cases in the test suite are used as T_g. Since different initial failed test cases may lead to different experimental results and in order to reduce the bias that different initial failed test cases may cause, every experiment is repeated 20 times. That is to say initial failed test cases are randomly chosen every time, and the final experimental results are averaged based on these experiments.

Table 1. Detailed information of the subject programs

Subject	Versions	LOC	Size of TS
print_tokenens	4	565	4130
print_tokens2	10	508	4115
replace	27	563	5542
schedule	4	368	2650
schedule2	8	307	2710
tcas	35	173	1608
tot_info	23	406	1052
flex	23	5217	567
grep	21	12653	809
gzip	15	6573	213

LOC: lines of code for each program, TS: total number of available test cases.

B. Evaluation Metrics

To verify the effectiveness of the proposed approach, it is expected that our approach should select a small portion of test cases for result inspection. Meanwhile, the selected test cases should ensure a certain level of fault-localization effectiveness. The following metrics are used in this paper.

1) Percentage of Test Cases Selected

$$\mathrm{SR} = \frac{|T_r|-1}{|T_g|+|T_r|-1}$$

Where $|T_r|-1$ denotes the number of selected test cases for result inspection, and $|T_g|$ denotes the number of test cases in T_g that are not selected. A smaller *SR* indicates that fewer test cases are chosen for result inspection.

2) Effectiveness of Fault Localization:

$$\mathrm{Expense} = \frac{|V_{examined}|}{|V|} \times 100\%$$

Where $|V|$ is the size of executable statements in the program P, and $|V_{examined}|$ is the number of statements that the developers have to check following suspicious rank generated by CBFL techniques in order to find the fault. A small value of *Expense* indicates that fault localization is effective. In this paper, an Expense reduction score $\Delta Expense = Expense - Expense'$ is also used to measure relative effectiveness improvement; where *Expense* refers to Expense value using the whole test cases for CBFL techniques while *Expense*' refers to *Expense* value using the

test cases selected by our approach. A positive value of $\Delta Expense$ indicates that the effectiveness of fault localization gets improved after applying our approach.

V. EXPERIMENTAL RESULTS AND ANALYSIS

In this section, several experiments are conducted to study the effectiveness of our proposed approach and the raw results of the experiments across the subject programs are shown in detail for further analysis. The following questions are investigated: 1) Can the proposed method help programmers select a few test cases for result inspection from a large number of test cases whose execution results are not available? 2) Can the test cases selected by the proposed approach effectively locate faults when some typical fault localization techniques are applied? 3) Compared with other techniques, is the proposed approach better than related works?

A. Percentage of Test Cases Selected

This section investigates whether our method can select a few test cases from a large number of test cases for result inspection. The experimental results are shown using a boxplot. Figs. 3(a) and (b) list the distribution of SR on *Siemens* and *Unix* programs, respectively.

From Fig. 3, it can be observed that the size of test cases selected by our method is much smaller than the size of the original test suite. Moreover, SR ranges from 0.12% to 2.9% for all faulty programs. For the same program, SR of different faulty versions fluctuates slightly. Take *replace* for example. SR varies in the range [0.27%, 1.24%], and SR of 88.9% faulty versions is smaller or equal to 1%. Furthermore, it can be seen that SR for *tot_info* programs is greater than those of other programs. It is because the size of the test suite in *tot_info* is smaller.

From Fig. 3(b), we can observe that most of SR in *Unix* programs is greater than 5%. One reason is that the sizes of the test suite in *Unix* programs are usually small. The other reason is that *Unix* programs are more complex than *Siemens* programs, hence more test cases are needed to divide as many groups as possible.

Fig. 4 lists the distribution of SR over all programs. The x-axis denotes the range of SR, and the y-axis denotes the percentage of program versions that satisfy the corresponding range of SR. It can be observed that SR of 64% faulty versions is smaller than 1.0%, of which 15.8% versions is smaller than 0.5%. In summary, it can be concluded that the proposed approach can significantly

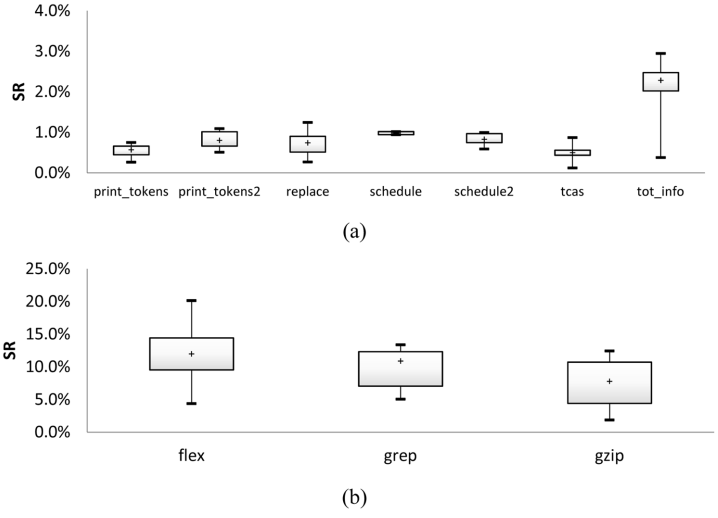

Fig. 3. Distribution of SR for subject programs: (a) *Siemens*, (b) *Unix*. SR: percentage of test case selected.

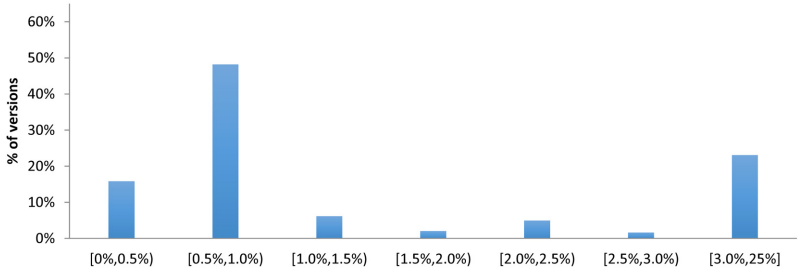

Fig. 4. Distribution of SR over all program. SR: percentage of test case selected.

reduce the test cases needed to be inspected by developers.

B. Effectiveness of Fault Localization

This section investigates whether the selected test cases can support effective fault localization. Fig. 5(a) and (b) list the distribution of ΔExpense on two typical fault localization techniques *Tarantula* and *Ochiai*, respectively. From these figures, it can be observed that the ΔExpense rises or falls around 0%, which indicates that chosen test cases in various faulty versions reveal various improvement of fault-localization effectiveness. Take *Tarantula* and *schedule2* for example. Note that more than half of the faulty versions in schedules reveal better fault localization than that when original test suites are used, which implies the effectiveness of fault localization can be improved using only a few test cases. It is because that research [20] indicates that redundancy may exist in a high-coverage test suite. The redundancy may introduce a negative impact on fault localization. Moreover, coincidentally correct test cases may also exist in test suite, which may introduce inverse impact on fault localization [21]. The method proposed in this paper only selects a few tests from the original test suite, which potentially reduce redundancy and coincidentally correct tests that may affect effectiveness of fault localization. Furthermore, the proposed method aims to select test cases that can distinguish statements in a program, thus it is reasonable that the approach can improve fault-localization effectiveness. Overall, by using our approach 54.7% faulty versions get improved or is close to that of the whole test suites when *Tarantula* is applied, 61% when *Ochiai* is applied. Based on Figs. 4 and 5, it is obvious that using the proposed method, developers need to check only a small portion of all the test cases, while the reduced test information can still support effective fault localization for most programs.

Furthermore, by comparing the effectiveness of fault localization on the same program versions using *Tarantula* or *Ochiai*, it is obvious that using the same chosen test cases as inputs, *Ochiai* is more effective in locating faults than *Tarantula*. Table 2 lists results of ΔExpense for two techniques on faulty versions of schedules. We can observe that among the versions in which fault localization improves, *Ochiai* is quicker than *Tarantula* at finding faults, such as versions v6 and v10. The same phenomenon can be also seen on the versions in which

Table 2. Fault-localization effectiveness on schedule2 programs

schedule2	Tarantula (%)	Ochiai (%)
v1	31.25	32.81
v2	-4.38	-0.78
v3	0	0
v4	0	0
v5	-13.07	-11.54
v6	13.95	19.38
v7	-4.30	-0.78
v10	0.76	10.50

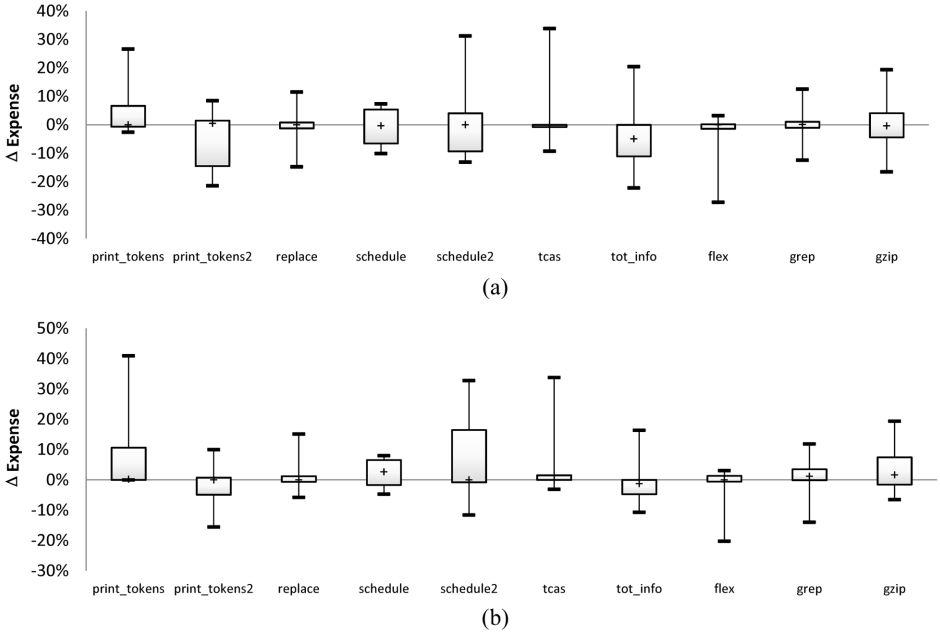

Fig. 5. Distribution of Expense reduction scores: (a) *Tarantula*, (b) *Ochiai*. ΔExpense: expense reduction score.

fault localization get deteriorated, such as versions v2 and v5. This is because compared with *Tarantula*, *Ochiai* gives more weight to the failed test cases in calculating the suspicion of statements, and thus it can locate faults effectively even when there is little test information. Therefore, when test information is limited, *Ochiai* behaves in more stable manner than *Tarantula* in locating faults. We recommend that when the method proposed in this paper is applied to fault localization techniques, a more effective fault localization should always be the first choice.

In summary, from the above experimental results, we can conclude that the proposed approach selects only a few test cases for result inspection, and meanwhile the selected test cases can support effective fault localization.

C. Comparison with Other Related Techniques

1) Comparison with Random Selection Method

Random selection method randomly selects some test cases from test suite as a reduced test set for each faulty version. To compare with the random selection method, suppose the method selects the same number of test cases as our proposed method does for each version. To reduce the bias that random selection may yield, for each version 20 random selections are made, and the averaged results are used as final results.

Fig. 6 lists the expense of fault localization yielded by two methods using two fault localization techniques, in which y-axis denotes averaged Expense for corresponding programs. From these figures, it can be observed that selecting the same number of test cases for fault localization, our method is more effective than random selection method at finding faults. Take *Tarantula* for example. In *replace* programs, the averaged Expense is 4.9% less than random method. Specifically, each faulty version of *replace* program contains about 244 executable code lines. That is to say, when the test cases selected by our method are used, on average developers check 12 code lines less than random selection method. In complex *Unix* programs, the improvement is much more obvious. In summary, when *Tarantula* is used, the Expense of our method is 6% less than random method on average. The same phenomenon also can be seen when *Ochiai* is used as the fault localization technique. It indicates our method can select test cases that are more conducive to fault localization than a random method.

2) Comparison with Peer Technique

To further illustrate the effectiveness of our approach, we compare our approach with peer work. Hao et al. [17] proposed three test case reduction strategies for fault localization. Among the techniques they proposed, the third strategy can assist fault localization techniques, locating more faults than the other two. Therefore, their third technique is used for comparison and is denoted as 'HAO-S3'. Our approach is denoted as 'EDS'. Table 3 lists experimental results of our approach and HAO-S3, respectively.

From Table 3, it can be seen that our approach can

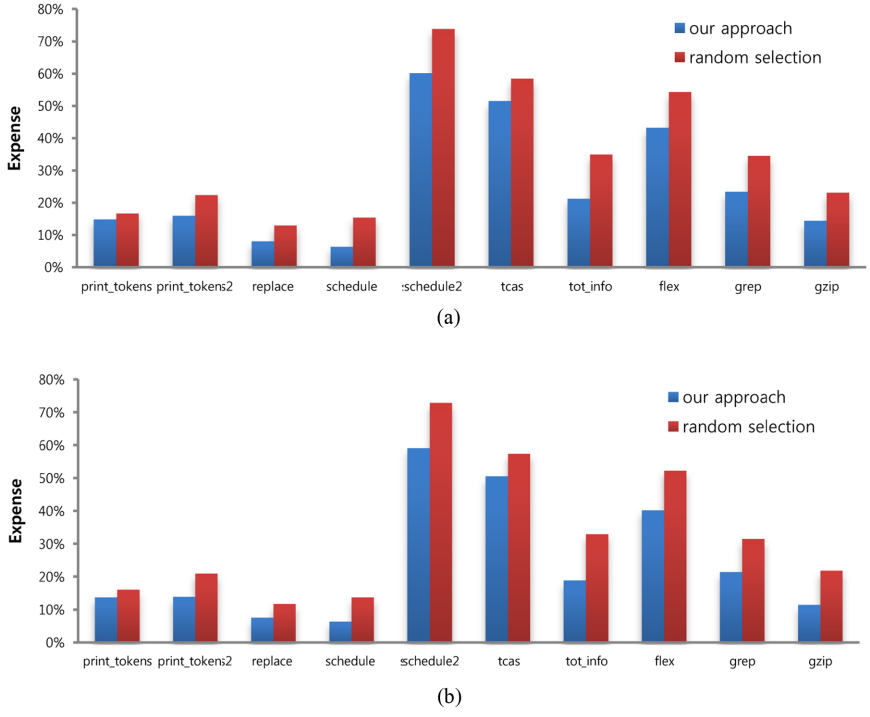

Fig. 6. Fault localization effectiveness: (a) *Tarantula*, (b) *Ochiai*. Expense: effectiveness of fault localization.

Table 3. Comparison of SR and Expense for different approaches

Program	Approach	SR (%)	Tarantula (%)	Ochiai (%)
print_tokens	EDS	0.54	14.8	13.7
	HAO-S3	0.71	12.0	12.8
print_tokens2	EDS	0.83	16.0	13.9
	HAO-S3	1.00	17.3	14.1
replace	EDS	0.71	8.1	7.5
	HAO-S3	1.17	9.4	7.8
schedule	EDS	0.98	6.3	6.4
	HAO-S3	2.63	7.7	7.7
schedule2	EDS	0.83	6.01	5.91
	HAO-S3	1.37	6.06	6.06
tcas	EDS	0.42	5.15	5.05
	HAO-S3	0.56	5.15	5.05
tot_info	EDS	2.02	21.3	18.9
	HAO-S3	2.35	23.5	20.2
flex	EDS	11.6	43.1	40.2
	HAO-S3	13.4	43.6	42.1
grep	EDS	9.92	23.5	21.4
	HAO-S3	11.3	24.1	22.4
gzip	EDS	7.51	14.6	11.5
	HAO-S3	8.67	15.0	13.1

SR: percentage of test case selected.

select fewer test cases for fault localization while the effectiveness of fault localization is close to or even improves over that of HAO-S3 on most of the programs. Specifically, Table 4 lists the experimental data on some faulty versions, in which *Tarantula* technique is applied. Take print_tokens2_v3 for illustration. Twenty-one test cases are selected for result inspection by our approach, and by Tarantula using the selected test cases the faulty statement ranks in at 9th place. Compared with our approach, HAO-S3 chooses 19 more test cases while the faulty statement is ranked at the 11th place.

To further investigate the difference on percentages of selected test cases for two approaches, we conduct Wilcoxon signed-rank test on all the subjects programs. Table 5 lists the p-value on individual program. It can be seen that all but print_tokens programs of the p-values are smaller than 0.05, which indicates the improvements are significant at the 5% level. Note that the data in Table 3 indicates that our approach selects fewer test cases than HAO-S3, thus it can be concluded that our approach can significantly reduce the test cases used for effective fault localization compared with HAO-S3. The reason is that our approach employs dynamic basic blocks to measure whether statements are distinguishable, and O^p metric is used to estimate the fault probability of groups. Moreover, groups and indistinguishable groups are used to assess division of test cases, which to some extent reduce redundant test cases while test cases that are helpful for fault localization are not discarded.

Table 5. p-value for programs

Program	p-value
print_tokens	6.25×10^{-1}
print_tokens2	3.85×10^{-2}
replace	3.38×10^{-5}
schedule	1.25×10^{-2}
schedule2	2.25×10^{-2}
tcas	2.14×10^{-3}
tot_info	1.24×10^{-2}
flex	2.13×10^{-2}
grep	4.12×10^{-2}
gzip	3.11×10^{-2}

VI. DISCUSSION

In this section, we move to further discussions on the approach proposed in the paper.

The approach proposed in this paper assumes that when a test case is chosen, the developers check the execution result of this test, and that the check is correct. However, since developers may have different under-

Table 4. Comparison on some faulty versions for two approaches

Faulty version	EDS		HAO-S3	
	No. of test cases	Rank of faulty statement	No. of test cases	Rank of faulty statement
print_tokens_v2	11	9	14	7
print_tokens2_v3	21	9	40	19
replace_v1	15	5	19	5
tcas_v37	11	2	11	8

standings of program specification, the check may not be always correct. In this paper, for simplified discussion on the effectiveness of our approach, we assume the check is always correct. Further investigation on the situation when the developers make mistakes in labeling the execution result of tests and its impact on fault localization will be conducted in the future.

When our approach plans to select one test case from T_g for inspection, the test case to be chosen is implicitly correlated with previous test cases selected. This is because our approach gives high priority to the test case that can divide the dynamic basic blocks that are covered by a high rate of failed test cases. However, our approach cannot guarantee that failed test cases are always selected out of T_g. Once the test suite T_r doesn't contain extra failed test cases, our approach tends to select test cases that can divide dynamic basic blocks as evenly as possible, which may not be effective for later fault localization. This is one disadvantage of our approach. In the future, we plan to use program slice to analyze the relations between statements and the initial failed test case, and use the fault proximity approach proposed by Liu et al. [22] to increase the probability of selecting failed tests further.

Finally, our approach can be applied to most of the coverage based fault localization techniques. This is because such techniques use the same inputs as the techniques applied in this paper to calculate the possibility of statements that contain faults. However, some fault localization techniques, e.g. [23, 24], assume the distribution of the coverage of statements on passed and failed test cases. Such assumptions require a large number of test cases with execution results available to support. However, our approach only selects a small portion of test cases that may not be sufficient to support the assumption. We will do more work in the future to study the commonality of our approach.

VII. CONCLUSION

In this paper, we propose a strategy to select test cases for result inspection from a large number of test cases without test oracles so as to support effective fault localization. DBBs are employed to represent each test case in *T*. Then groups and indistinguishable groups are introduced to gather dynamic basic blocks. Our approach tends to select the test case that can divide the groups evenly as well as cover dynamic basic blocks that potentially contain faulty statements. Once a test case in T_g can no longer divide the indistinguishable groups, the selection procedure terminates. The selected test cases together with the initial failed test case are then used for fault localization. To evaluate the effectiveness of our approach, experiments are conducted using *Siemens Test Suites* and *Unix* programs. Experimental results show that our approach can significantly reduce the test cases required for result inspection while the effectiveness of fault localization is close to or even improves over that of all the test cases. Experiments on comparison with related work also indicate our approach is more effective.

ACKNOWLEDGMENTS

The paper is supported by the State High-Technology Development Project of China with Grant No. 2013AA011701.

REFERENCES

1. I. Vessey, "Expertise in debugging computer programs: a process analysis," *International Journal of Man-Machine Studies*, vol. 23, no. 5, pp. 459-494, 1985.
2. J. A. Jones, M. J. Harrold, J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, Orlando, FL, 2002, pp. 467-477.
3. R. Abreu, P. Zoeteweij, and A. J. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proceedings of 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, Riverside, CA, 2006, pp. 39-46.
4. L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 3, article no. 11, 2011.
5. J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, Long Beach, CA, 2005, pp. 273-282.
6. S. Ali, J. H. Andrews, T. Dhandapani, and W. Wang, "Evaluating the accuracy of fault localization techniques," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering,* Auckland, New Zealand, 2009, pp. 76-87.
7. C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, Toronto, Canada, 2011, pp. 199-209.
8. Y. Yu, J. A. Jones, and M. J. Harrold, "An empirical study of the effects of test-suite reduction on fault localization," in *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, 2008, pp. 201-210.
9. P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, 2005, pp. 213-223.
10. K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering,* Lisbon, Portugal, 2005, pp. 263-272.

11. E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi, "Toward the determination of sufficient mutant operators for C," *Software Testing, Verification and Reliability*, vol. 11, no. 2, pp. 113-136, 2001.
12. X. Xie, W. E. Wong, T. Y. Chen, and B. Xu, "Metamorphic slice: an application in spectrum-based fault localization," *Information and Software Technology*, vol. 55, no. 5, pp. 866-879, 2013.
13. A. Gonzalez-Sanchez, E. Piel, H. G. Gross, and A. J. Van Gemund, "Prioritizing tests for software fault localization," in *Proceedings of 2010 10th International Conference on Quality Software (QSIC),* Zhangjiajie, China, 2010, pp. 42-51.
14. A. Gonzalez-Sanchez, R. Abreu, H. G. Gross, and A. J. Van Gemund, "Prioritizing tests for fault localization through ambiguity group reduction," in *Proceedings of 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE),* Lawrence, KS, 2011, pp. 83-92.
15. L. Gong, D. Lo, L. Jiang, and H. Zhang, "Diversity maximization speedup for fault localization," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Essen, Germany, 2012, pp. 30-39.
16. S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, Trento, Italy, 2010, pp. 49-60.
17. D. Hao, T. Xie, L. Zhang, X. Wang, J. Sun, and H. Mei, "Test input reduction for result inspection to facilitate fault localization," *Automated Software Engineering*, vol. 17, no. 1, pp. 5-31, 2010.
18. Y. H. Li and L. L. Yu, "A test case selection method to support fault localization," in *Proceedings of 2014 International Conference on Computer Science and Software Engineering (CSSE2014)*, Hangzhou, China, 2014, pp. 129-135.
19. B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," in *Proceedings of the 28th International Conference on Software Engineering*, Shanghai, China, 2006, pp. 82-91.
20. D. Hao, L. Zhang, Y. Pan, H. Mei, and J. Sun, "On similarity-awareness in testing-based fault localization," *Automated Software Engineering*, vol. 15, no. 2, pp. 207-249, 2008.
21. W. Masri, R. Abou-Assi, M. El-Ghali, and N. Al-Fatairi, "An empirical study of the factors that reduce the effectiveness of coverage-based fault localization," in *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, Chicago, IL, 2009, pp. 1-5.
22. C. Liu, X. Zhang, and J. Han, "A systematic study of failure proximity," *IEEE Transactions on Software Engineering*, vol. 34, no. 6, pp. 826-843, 2008.
23. C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "SOBER: statistical model-based bug localization," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 286-295, 2005.
24. Z. Zhang, W. K. Chan, T. H. Tse, Y. T. Yu, and P. Hu, "Non-parametric statistical fault localization," *Journal of Systems and Software*, vol. 84, no. 6, pp. 885-905, 2011.

Yihan Li

Yihan Li was born in 1985. He received his Ph.D. degree from School of Computer Science and Engineering, Beihang University, China in 2015. His main research interests include software testing and software debugging.

Jicheng Chen

Jicheng Chen was born in 1976. Ph.D., associate professor, Member of China Computer Federation. His main interests include computer architecture, cache coherence and software testing.

Fan Ni

Fan Ni was born in 1984. Ph.D. His main research interests include computer architecture, real time system and software testing.

Yaqian Zhao

Yaqian Zhao was born in 1981. Ph.D. Her main research interests include software reliability and machine learning.

Hongwei Wang

Hongwei Wang was born in 1982. Ph.D. His main research interests include high performance computing and artificial intelligence.