Effective Fault Localization Using Weighted Test Cases

Yihan Li, Chao Liu
School of Computer Science and Engineering
Beihang University
Beijing, China
Email: {liyihannew, liuchao}@sei.buaa.edu.cn

Abstract—Locating faults in a program is prohibitively time-consuming and tedious, and therefore, many automated fault localization techniques have been proposed to assist in the debugging process. Spectrum based fault localization are promising techniques that can guide developers to the possible locations of faults. These techniques make a summary on the number of passing and failing tests cases to prioritize suspicious statements according to likelihood of containing program bugs for each statement. Though results are encouraging, these techniques treat all test cases as equally important, which ignore individual error diagnosis ability for different test cases. In this paper, we present an approach to exploit varying weights for individual test cases in the computation of suspiciousness scores so as to improve the effectiveness of spectrum based fault localization techniques. To validate our method, experiments were performed on eight typical SFL techniques using two standard benchmarks. Results are suggestive that for the studied SFL techniques, our method can significantly improve the fault localization effectiveness in most situations and in other cases it does not introduce much adverse impact on the techniques original performance.

Index Terms—fault localization, program spectrum, debugging, passed tests, failed tests

I. INTRODUCTION

It is a common phenomenon that no matter how much effort developers spend on testing a program, software faults are introduced and removed continually during software development processes. When faults are revealed during testing processes, software debugging is often used to remove as many faults in the program as possible so as to improve the quality of the program. Typically, software debugging involves locating faults, repairing faults and verifying repairs. Among the various debugging activities, locating faults has been recognized as one of the most expensive [1]. Therefore, over the last ten years there has been an explosion of work in automatic fault localization techniques [2]–[8] that assist developers in finding the locations of faults, thereby alleviating the work developers devoted in debugging.

In particular, Spectrum-base fault localization (SFL) is a low-cost and effective technique that tries to pinpoint the possible locations of faults. Essentially, SFL aims to correlate program entities with program failures via statistically analyzing coverage information. Specifically, SFL collects coverage information from a set of test cases together with their corresponding test results to form program spectrum, and then contrasts the coverage statistics of program entities between passed runs and failed runs using various statistical formulas. Finally, it yields a ranking list that rank all program entities in terms of suspiciousness. The program entities with higher suspicious value are more likely to contain bugs and thus are given higher priority for examination during software debugging. Based on coverage information, researchers have proposed various statistical formulas to measure the correlation between program entities and program failures, such as Tarantula [9], Ochiai [10] and Jaccard [10].

Although SFL approaches have brought encouraging results in locating faults, most of SFL approaches do not distinguish the contribution of test cases from each other. Basically, these approaches take information of the number of failed and passed tests into account and thus assume all test cases share equal importance, which ignore individual fault detection ability for different test cases and may limit effectiveness of fault localization. Take Tarantula as an example. Suppose that two statements are both exercised by the same number of passed tests and that of failed tests while are executed by different tests. In such case, the technique assigns same suspiciousness score to these two statements and thereby loses ability to distinguish these two statements in the ranking list, which may decrease its accuracy in predicting the locations of bugs. Furthermore, during testing, the distribution of passed tests and failed tests are always uneven. That is, the number of failed tests is often relatively smaller than that of passed tests. While SFL assigns same weights to all tests, the contribution of the faulty statement to failure will also be decreased largely in the case that the faulty statement is executed by relatively more passed tests, which may make the faulty statement ranked lower than some other non-faulty statements in the suspicious list.

In this paper, we extend our previous study in [11] and propose a weighting strategy to measure contribution of different test cases so as to mitigate noise induced by tests. Our method distinguishes the weight of one failed test case from another or one passed test case from another. For each failed test, different weights are assigned with respect to each statement according to its coverage information and proportion between failed tests and passed
tests. For each passed test, weights are assigned according to
its average similarity to all failed tests. In such a
way, our method can be generally applied to existing
SFL techniques, such as Tarantula et al., to improve fault
localization effectiveness. We use Siemens programs and
three Unix programs as our subject programs to evaluate
our strategy, and compare performance of several fault
localization techniques using the conventional formulas
and our refined formulas respectively. The empirical re-
results show that our method is promising on the studies
subject programs. Further analysis shows that for the
studied metrics, our approach can significantly improve
their performance in locating faults. The contributions of
this paper can be summarized as:

1) We propose an approach to quantify weights for
failed tests and passed tests respectively according
to coverage information so that statistics of data
from tests is enriched which can benefit fault lo-
calization. Our approach can be easily applied to
existing statistical formulas without much modifi-
cation.

2) We evaluate the effectiveness of our weighting
approach systematically across two standard bench-
marks and compare performance of several SFL
techniques using the conventional method and our
proposed method. Comparisons are also made with
peer woks.

The rest of the paper is organized as follows. Section
II introduces some related works in fault localization.
Section III presents our techniques. We detail our weight-
ing scheme in this section. Section IV providers our
experiment and analysis. Finally, Section V concludes our
work and presents future work.

II. RELATED WORKS

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

Agrawal et al. [12] are the first to propose a coverage
based fault localization technique called dicing, which
subtracts the set of statements executed by a passed test
case from those executed by a failed one and reports the
result statements as the likely faulty method. Renieris
et al. [4] extended this idea and proposed Nearest Neigh-
borhood Queries technique. The technique compares the
spectra of the successful runs and failed runs, then select
the nearest passed run as one of input to dicing.

The Tarantula system [9] colored the statements to
highlight the particular statements that contain bugs. It
used the number of successful tests and failed tests to
locate buggy statements. The intuition is that the more
failed tests and the less successful tests cover a state-
ment, the more likelihood for the statement to be faulty.
Different colors for different suspicious statements are
then assigned to visualize program codes. Therefore a
buggy statement is colored as red for highlighting so that
developers can focus on it directly. Empirical evaluation
[13] showed that Tarantula consistently outperforms four
other techniques: Set union [4], Set intersection [4], Nearest
Neighbor [4], and Cause Transitions [14].

Abreu et al. [10] proposed several spectra metrics to
study the accuracy of prediction by fault localization. In
their work, they found that Ochiai metric is more effective
in bug localization performance than other metrics. Simi-
lar to Tarantula, these metrics only use binary information
of test execution to rank statements.

Wong et al. [15] proposed some heuristic strategy
to assign different weights for passed and failed tests
respectively. Weights for passed/failed tests are assigned
according to the number of passed/failed tests. The tests
are grouped and tests in the different group are assigned
different weights. As the total number of successful tests
and the total number of failed tests are fixed, weights
assigned to tests are fixed regardless of individual ex-
ecution information. Moreover, their method does not
distinguish weights for individual tests in the same group
that contribute to statements.

Naish et al. [5] proposed a weighting strategy for failed
tests. The rationale behind the idea is that failed tests
that cover few statements provide more information than
other failed tests. Thus, they assumed that weight of a
failed test is inversely proportional to the number of
statements exercised in the test. Inspired by their work,
we will extend their studies in three aspects: 1) We use
basic blocks rather than statements to calculate weight
for each failed test since statements within a basic blocks
are often not distinguishable from each others in terms of
error diagnosis. 2) We additionally consider the imbalance
property of tests with respect to each statement. The
weight of a failed test is assigned according to proportion
between the number of failed tests and that of passed
tests. 3) Weights of passed tests are also quantified. In
this study, we also compare our weighting strategy with
method proposed by Naish et al.

Bandyopadhyay et al. [6] extended the idea of nearest
neighbor queries [6] to incorporating the relative impor-
tance of different passing test cases in the calculation of
suspiciousness scores. They stated that the importance of
a passing test case is proportional to its average proximity
to the failing test cases. They used different thresholds for
their weighting function to control weights assigned to
tests. However, in their study, they do not prescribe how
to choose best threshold for weighting function. Different
from their work, our work focuses on relative importance
of failing tests and passing tests.

III. APPROACH

A program spectrum is a collection of data that record
the statements that are executed in each test case. To sum
up such information, the current popular SFL techniques
utilize four spectrum parameters. They are the number
of passed/failed test cases in which statement was/wasnt
executed. Following Abreu et al. [10], the notation
\( \alpha_{cf}(s), \alpha_{nf}(s), \alpha_{op}(s), \alpha_{np}(s) \) is used to denote these
four numbers for each statement \( s \) to calculate suspicious-
ness score based on statistical formula. The first part of
the subscript in the notation for each parameter indicates whether the statement is executed (e) or not (n) and the second one indicates whether the test passed (p) or failed (f). In the context of clarity that these spectrum parameters are provided for the specific statement s, we sometimes omit the notation "(s)" in the spectrum parameters for simplicity. Similar to this, in this paper, four weighted spectrum parameters with respect to each statement are defined as follows:

\[ N_{ef}(s) \text{: the weights of failed test cases that cover statement s} \]
\[ N_{nf}(s) \text{: the weights of failed test cases that do not cover statement s} \]
\[ N_{sp}(s) \text{: the weights of successful test cases that cover statement s} \]
\[ N_{np}(s) \text{: the weights of successful test cases that do not cover statement s} \]

The contribution for different types of tests to failure is considered separately in terms of error diagnosis. The reason is that typically failed tests present definite information about program behavior that fault is activated and propagated to program failure, while passed tests do not guarantee that fault is activated. The computation of the above four weights for each statement will be described in the following sections. To facilitate the discussion in the rest of the paper, let us suppose that a program \( P \) consists of \( n \) executable statements, which are denoted as \( P = \{s_1, s_2, s_3, \ldots, s_n\} \), and \( m \) basic blocks, which is denoted as \( P = \{b_1, b_2, b_3, \ldots, b_m\} \). Also consider that the program \( P \) is tested against a test suite \( T \), which comprises of \( w \) different test cases that are denoted as \( T = \{t_1, t_2, t_3, \ldots, t_w\} \). The test suite \( T \) can be partitioned into two disjoint subsets \( T_p \) and \( T_f \) according to the passed/failed status of test cases.

A. Weights of Failed Tests

In this section, we describe how to compute \( N_{ef}(s) \) and \( N_{nf}(s) \) for each statement in detail. Two practical test scenarios that motivate us to distinguish weights for failed tests are discussed respectively, and some equations are defined accordingly to capture characteristic of weights induced by test scenarios. Then these equations are combined properly to measure different weights of failed tests with respect to each statement.

1) Weight of coverage of test with respect to each statement: Suppose two tests both trigger program failure while one of them executes less basic blocks than the other does. When measuring fault locating ability for each test, it seems that the test that exercises less basic blocks gives us more information to find fault than the other one, since less basic blocks narrow down the search space for possible locations of faults. In the extreme case that the test executes only one basic block, it is certain that the executed basic block contains faulty statements. The conventional SFL techniques only use the number of failed tests in which a statement is executed, which lose such information. Note that here basic blocks are used rather than statements to capture this characteristic since statements within the same basic block cannot be distinguished by tests.

Let a test case \( t_i \) be one element of \( T_f \) and cover statement \( s \). Spectra of basic block information for \( t_i \) are collected, which is recorded as a binary vector \(< b_{i1}, b_{i2}, b_{i3}, \ldots, b_{imi} >\). If block \( b_j \) is covered by test \( t_i \), then the value for \( b_{ij} \) in the vector is 1 otherwise 0. After collecting basic block information, we compute weight of test case \( t_i \) about its execution that contributes to statement \( s \) using equation 1:

\[ E_{ci}(s) = m \sum_{r=1}^{m} b_{ri} \]  

(1)

The weight is inversely proportional to the number of basic blocks executed in the failed test. The greater the \( E_{ci} \) is, the more information the test case provides for locating faults. The weight approaches maximum when the failed test executes only one basic block. In such scenario, developers can immediately find the location of fault with the aid of this failed test. When a failed test executes all the basic blocks in the program, it provides relatively less information to aid in locating faults.

The weight for a statement \( s \) that is not executed by a failed test \( t_i \) is also quantified in the similar way under the assumption that a large value implies that the statement may not be correlated with program failure. The following equation computes such contribution of tests to the statements.

\[ E_{mi}(s) = m/(m - \sum_{r=1}^{m} b_{ri} + 0.01) \]  

(2)

A small constant (0.01 in this study) is used in the denominator to avoid division by zero when all of the basic blocks are executed in a failing test. The weight quantifies the degree that the failed test subjects the statement to be a non-faulty one.

2) Weight of status of tests for failed tests: In testing process, the number of failed tests is often relatively smaller than that of passed tests. It indicates that failed tests are often rarer than passed tests in test sets. Furthermore, a failed test definitely tells us that there exist some faults in the program while a passed test does not. This implies that contribution of failed tests to fault localization should be distinguished from that of passed tests. The conventional SFL techniques do not make full use of this feature, which decrease the effect of failed tests for error diagnosis and may degrade effectiveness of SFL. In this paper, we propose an information quantity [16] based strategy to reduce imbalance between sizes of passed and failed tests, thus contribution of failed tests to failure is dynamically determined by proportion between failed and passed tests. Specifically, let an event be “The test set contains \( (h+k) \) tests, of which the number of failed tests is \( k \).” We assume that occurrence of test cases is stochastic and independent from each other. The probability for occurrence of this event equals to \( k/(h+k) \). Thus, the information quantity for the event can be represented as:

\[ -\log\left(k/(k + h)\right) = \log\left((h + k)/k\right) \]
measures how much information is contained by an event. A smaller probability this event occurs with a larger information quantity the event has. Motivated by this property, we use the information quantity of the event as weight for every k failing tests. Therefore, the weight of every failed test \( t_i \) can be computed by equation 3:

\[
I = \log((h + k)/k + 1)
\]  

(3)

A constant (1 in this study) is added in the equation to ensure that weight of a failing test is always greater than 1. The formula dynamically determines weight for every failed test according to size relation between passed tests and failed tests. A great value indicates that the failed information is rare for fault localization and weight of failed tests are adjusted for better error diagnosis. Such case often happens when many passed test cases and a few failed test cases are contained in the test suite, which is common in test process. In such cases, passed test cases may be already redundant while failed test cases are lacking for locating faults, which implicitly indicate that we should pay relatively more attention to such failed test cases owing to redundancy of passing tests.

Next, we quantify the contribution of failed tests associated with statements. Suppose a statement \( s \) is executed by a failed test \( t_i \). The statement gets more suspicious as \( a_{e_i} \) increases. Our intuition is that the contribution of a failed test to a statement should be increased as the statement is executed by more failed tests. To characterize such contribution, we use the following equation to compute:

\[
RC(s) = I \times \log(a_{e_i}/(|T_f| - a_{e_i} + 0.1) + 1)
\]  

(4)

The constant 0.1 is used to avoid division by 0. Similarly, the weight of the failed test \( t_i \) for a statement \( s \) that is not executed by \( t_i \) is calculated using equation 5:

\[
RN(s) = I \times \log(a_{e_i}/(|T_f| - a_{e_i} + 0.1) + 1)
\]  

(5)

3) Combined to measure weights of failed tests: To compute \( N_{ef}(s) \) and \( N_{nf}(s) \) of failed tests with respect to every statement in the program, the above four equations are combined as follows.

For a statement \( s \) that is executed by a failed test \( t_i \), the contribution of this test to the statement is computed as: \( E_{ci}(s) \times RC(s) \). Thus, to compute weights of failed test set \( F \) all of which covers the statement \( s \), we sum over weight of each test case in \( F \) using equation 6.

\[
N_{ef}(s) = \sum_{t_i \in F} E_{ci}(s) \times RC(s)
\]  

(6)

Similarly, for a failing test \( t_i \) that does not cover a statement \( s \), the weight of the test to the statement is computed as: \( E_{ni}(s) \times RN(s) \). Therefore, eq. 7 is used to compute weights of a failed test set \( U \) all of which do not cover the statement \( s \) by summing over weight of each test case in \( U \).

\[
N_{nf}(s) = \sum_{t_i \in U} E_{ni}(s) \times RN(s)
\]  

(7)

B. Weights of Passed Tests

Passed tests provide clues about whether the executed statements are innocent of program failures or not. However, passed tests may also execute faulty statements and recent studies revealed that fault localization is susceptible to such tests [10]. The extents to which passed tests subject statements not to be faulty ones may vary for various passed tests due to their execution information. For example, if statements executed by a specific passed test share little intersection with statements of other failed tests, we have high confident to conclude that the statements executed by the passed test may be free from program faults and thus give those statements lower examination priority. However, the current SFL does not distinguish contribution of passed tests. To mitigating negative impacts of such tests on fault localization, we proposed a method to quantify weights of passed tests according to how close the passing test is to failed tests.

Suppose a test case \( t_i \) is denoted as a binary vector \( < e_{i1}, e_{i2}, \ldots, e_{in} > \), where \( e_{ij} = 1 \) indicates the statement \( s_j \) is executed by the test while \( e_{ij} = 0 \) indicates the statement is not executed. To quantify how similar two tests are, a modified Ochiai similarity coefficient [17] is used:

\[
\text{Sim}(t_i, t_j) = \frac{v(t_i \cap t_j)}{\sqrt{v(t_i) \times v(t_j)}}, \text{where}
\]

\[
v(t_k) = \sum_{m=0}^{n} I_{km} \text{ and } I_{km} = \begin{cases} 
1 & \text{if } e_{km} = 1 \\
0 & \text{others}
\end{cases}
\]  

(8)

The function of \( v(t_k) \) measures the closeness of \( t_k \) to failed tests. We assign various contributions of statements to failure according to execution information. The greater the \( \text{Sim}(t_i, t_j) \) is, the more similar two tests are. If two tests executed the same statements, the similarity between them approximate maximum, that is 1. If two tests shared no same statements in their executions, the similarity between them is 0. Based on this, we assign a lower weight to a passed test that is more similar to the failed test. Thus, given a set of failed tests, the weight for a passed test \( t_p \) is computed as average similarity with failed tests using following equation:

\[
W(t_p) = \frac{\sum_{t_i \in T_f} (1 - \text{Sim}(p, t_i))}{|T_f|}
\]  

(9)

For every statement \( s \) and a passed test set \( C \) all of which covers the statement \( s \), the weights of test set \( C \) for statement \( s \) are recorded in \( N_{ep} \), which is summed over each test case in \( C \) as equation 10:

\[
N_{ep} = \sum_{t \in C} W(t)
\]  

(10)
TABLE I.
STATISTICS OF SUBJECTS PROGRAMS

<table>
<thead>
<tr>
<th>Subject</th>
<th>Faulty Versions</th>
<th>LOC</th>
<th>Size of Test Pool</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>4</td>
<td>565</td>
<td>4130</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>10</td>
<td>510</td>
<td>4115</td>
</tr>
<tr>
<td>replace</td>
<td>27</td>
<td>563</td>
<td>5542</td>
</tr>
<tr>
<td>schedule</td>
<td>4</td>
<td>412</td>
<td>2650</td>
</tr>
<tr>
<td>schedule2</td>
<td>8</td>
<td>307</td>
<td>2650</td>
</tr>
<tr>
<td>tcas</td>
<td>35</td>
<td>173</td>
<td>1608</td>
</tr>
<tr>
<td>todinfo</td>
<td>23</td>
<td>406</td>
<td>1054</td>
</tr>
<tr>
<td>flex</td>
<td>27</td>
<td>5217</td>
<td>567</td>
</tr>
<tr>
<td>grep</td>
<td>23</td>
<td>12653</td>
<td>809</td>
</tr>
<tr>
<td>gzip</td>
<td>17</td>
<td>6373</td>
<td>213</td>
</tr>
</tbody>
</table>

Similarly, for every statement s and a passed test set N all of which does not covers the statement s, a variable \( N_{np} \) is recorded, which is summed over each test case in N as equation 11:

\[
N_{np} = \sum_{t \in N} W(t) \tag{11}
\]

IV. EXPERIMENTS

In this section, we conduct two sets of experiments to evaluate the effectiveness of our algorithms. We compare the conventional method with our weighted method and peer works across subject programs.

A. Subject Programs

In this paper, Siemens suite and Unix programs are used as subject programs for the empirical studies, which are obtained from the Software artifact Infrastructure Repository [18]. These programs have been used to measure the effectiveness of fault localization techniques in previous studies [10], [13]. The Siemens suite contains seven small programs while Unix contains three relatively big programs. For each program, there are a variety of test cases and faulty versions available. Table I presents the detailed information on the subject programs. The Faulty 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 Test Pool represents the total number of available test cases in the test pool for each program.

Following previous work [10], [13], we excluded those faulty versions whose faults cannot be detected by any test case in the test suites, since failed runs are required for SFL techniques. Besides, we also remove the versions whose faults are introduced in non-executable statements, such as modifications in the header files, mutants in variable declaration statements, or modifications in a macro statement started with ‘#define’. We use the standard coverage tool gcov in conjunction with gcc to collect coverage information of program executions, and hence we also excluded those versions that gcov cannot handle owing to segmentation faults. In summary, we used all the remaining 143 faulty versions in our data analysis.

B. Formulas under investigation

After the above four weights are collected for each statement, the faulty statements are supposed to have relatively high \( N_{ef} \) values and relatively low \( N_{ep} \). To quantify how each program statement is correlated with program failure, some function that maps the four weights to a single value (we call such number suspiciousness score) for each statement can be applied to rank the statements. The property of employed function must satisfy the condition that those statements with the highest values are most possible to be faulty. In this paper, we apply our method to some SFL techniques as a refinement to calculate suspiciousness score for every statement. Table II shows eight conventional SFL techniques. Of the investigated techniques, Tarantula is an old technique and locates faults based on the assumption that the faults are generated from conventional formulas such as modulo. Thus, no adaptation is needed for use with non-integral values; The spectrum parameters \(< a_{ef}, a_{nf}, a_{ep}, a_{np} \) in conventional formulas can be simply substituted by \(< N_{ef}, N_{nf}, N_{ep}, N_{np} \) respectively to generate weighted based techniques. Of the investigated techniques, Tarantula is an old technique and locates faults based on the assumption that the faults are executed by relatively more failed tests and less passed tests [9]. Jaccard is used in the Pinpoint framework and evaluated very effective in previous studies [10]. Ochiai is often used for computing genetic similarity in molecular biology and is first used as fault localization technique in [19]. Ochiai is evaluated very effective in previous studies [10]. Ochiai is defined as:

\[
O(ch) = \frac{a_{ef} - a_{ep}}{a_{ef} + a_{ep} + a_{nf} + a_{np}} \tag{11}
\]

\[
O^n(ch) = \frac{a_{ef} - a_{ep}}{a_{ef} + a_{ep} + a_{nf} + a_{np}} \tag{12}
\]

\[
O^W(ch) = \frac{a_{ef} - a_{np}}{a_{ef} + a_{np} + a_{nf} + a_{np}} \tag{13}
\]

\[
O^W^n(ch) = \frac{a_{ef} - a_{np}}{a_{ef} + a_{np} + a_{nf} + a_{np}} \tag{14}
\]

\[
O^W(pch) = \frac{a_{ef} - a_{np}}{a_{ef} + a_{np} + a_{nf} + a_{np}} \tag{15}
\]

C. Evaluation Metric

To measure the effectiveness of fault localization techniques, we follow [20] to use Expense metric. It measures the percentage of the program that must be examined to find the fault following rank list from top down. The lower the measure is, the better the effectiveness is. It is defined as:

\[
\text{Expense} = \frac{\text{tot} \times \text{fail}}{\text{tot} \times \text{fail} + \text{print} \times \text{print} + \text{grep} \times \text{grep}} \tag{11}
\]

TABLE II.
REFINED FAULT LOCALIZATION TECHNIQUES

<table>
<thead>
<tr>
<th>Name</th>
<th>Formula</th>
</tr>
</thead>
</table>
| Tarantula  | \[
\frac{\frac{a_{ef} - a_{nf}}{a_{ef} + a_{nf} + a_{ep} + a_{np}}}{\frac{a_{ef} + a_{ep} + a_{nf} + a_{np}}{a_{ef} + a_{nf} + a_{ep} + a_{np}}} + \frac{a_{nf} + a_{np}}{a_{ef} + a_{nf} + a_{ep} + a_{np}}\]
| Jaccard    | \[
\frac{a_{ef} - a_{ep}}{a_{ef} + a_{ep} + a_{nf} + a_{np}} \tag{11}
\]
| Ochiai     | \[
\frac{a_{ef} - a_{ep}}{a_{ef} + a_{ep} + a_{nf} + a_{np}} \tag{11}
\]
| SFI        | \[
\frac{a_{ef} - a_{ep}}{a_{ef} + a_{ep} + a_{nf} + a_{np}} \tag{11}
\]
| O^n(ch)    | \[
\frac{a_{ef} - a_{ep}}{a_{ef} + a_{ep} + a_{nf} + a_{np}} \tag{11}
\]
| O^W(ch)    | \[
\frac{a_{ef} - a_{np}}{a_{ef} + a_{np} + a_{nf} + a_{np}} \tag{11}
\]
| O^W^n(ch)  | \[
\frac{a_{ef} - a_{np}}{a_{ef} + a_{np} + a_{nf} + a_{np}} \tag{11}
\]
| O^W(pch)   | \[
\frac{a_{ef} - a_{np}}{a_{ef} + a_{np} + a_{nf} + a_{np}} \tag{11}
\]

\[
W_{np} = \sum_{t \in N} W(t) \tag{11}
\]
as follow:

\[
\text{Expense} = \frac{|V_{\text{examined}}|}{|V|} \times 100\%
\]

|V| measures the size of executable codes in the program, and |V_{examined}| measures the number of statements that has to be inspected so as to find the fault. In case of tie when two or more statements share same suspiciousness score, we adopt the worst cases. That is, developers have to examine all the tie statements to find the faulty statement. In our study, Expense reduction score \(\Delta \text{Expense} = \text{Expense} - \text{Expense}'\) is also used to measure relative effectiveness improvement, where \(\text{Expense}\) and \(\text{Expense}'\) refers to Expense before and after applying our approach respectively. A positive value of \(\Delta \text{Expense}\) indicates that the effectiveness of fault localization is improved after applying our approach. The greater the measure value is, the more improvement the effectiveness of fault localization gains.

D. Results for comparison with conventional techniques

In this section, we present experimental results to investigate effectiveness in fault localization by applying the refinement method to all the formulas listed in Table II. Fig. 1 illustrates effectiveness between conventional SFL techniques and corresponding refined techniques in all faulty versions. For all the listed figures, the x-axis represents the percentage of executable statements to be examined, and the y-axis denotes the percentage of faulty versions whose faults have been located by examining no more than corresponding percentage of executable statements in the x-axis.

As shown in Fig. 1, the curves for eight revised formulas are always higher than the corresponding techniques. That indicates that all the eight revised formulas using our approach achieve better performance than, or at least competitive with the original SFL techniques in terms of effectiveness of fault localization. For example, based on Figure 1. (c), we find that by examining less than 10% of the code, the revised Wong2 formula can locate faults for
For most programs and techniques the reduction score for certain techniques may be that the faulty statements introduce much inverse impact for fault localization. That is, these techniques can still perform competitively for these techniques, our method does not introduce much inverse impact for fault localization. That is, these techniques can still perform competitively with original techniques.

For a more detailed comparison, Table III presents the mean expense reduction scores for eight refined techniques on each program. It can be seen from this table that for most programs and techniques the reduction score is always greater than 0%. This indicates that the effectiveness of conventional SFL techniques get improved by our weighting method in most cases. Among all the improvement, the highest one is in program tot_info using formula Wong2. The relative improvement is as high as 36.10%. Besides, we can also observe that for some techniques on certain programs, the relative improvement decrease. For example, the improvement on the program grep using refined Tarantula formula decreases by 0.90%.

TABLE III. MEAN EXPENSE REDUCTION SCORE FOR INDIVIDUAL PROGRAM

<table>
<thead>
<tr>
<th>Subjects</th>
<th>Tarantula (Revised)</th>
<th>Jaccard (Revised)</th>
<th>Ochiai (Revised)</th>
<th>SBI (Revised)</th>
<th>Wong2 (Revised)</th>
<th>Wong3 (Revised)</th>
<th>O (Revised)</th>
<th>O(^{P}) (Revised)</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>3.85%</td>
<td>3.21%</td>
<td>0.00%</td>
<td>5.31%</td>
<td>29.29%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>0.59%</td>
<td>12.42%</td>
<td>7.77%</td>
<td>12.42%</td>
<td>25.63%</td>
<td>0.39%</td>
<td>0.29%</td>
<td>0.29%</td>
</tr>
<tr>
<td>replace</td>
<td>0.29%</td>
<td>2.93%</td>
<td>0.70%</td>
<td>3.16%</td>
<td>28.74%</td>
<td>0.11%</td>
<td>-0.03%</td>
<td>-0.05%</td>
</tr>
<tr>
<td>schedule</td>
<td>0.67%</td>
<td>4.03%</td>
<td>1.51%</td>
<td>4.53%</td>
<td>9.24%</td>
<td>0.17%</td>
<td>0.17%</td>
<td>0.17%</td>
</tr>
<tr>
<td>schedule2</td>
<td>3.57%</td>
<td>11.82%</td>
<td>7.91%</td>
<td>12.22%</td>
<td>23.50%</td>
<td>0.05%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>tcas</td>
<td>0.22%</td>
<td>3.16%</td>
<td>1.54%</td>
<td>2.68%</td>
<td>19.52%</td>
<td>0.66%</td>
<td>0.09%</td>
<td>0.09%</td>
</tr>
<tr>
<td>tot_info</td>
<td>1.50%</td>
<td>8.84%</td>
<td>5.88%</td>
<td>10.16%</td>
<td>36.10%</td>
<td>0.46%</td>
<td>0.29%</td>
<td>0.29%</td>
</tr>
<tr>
<td>flex</td>
<td>0.33%</td>
<td>0.21%</td>
<td>0.20%</td>
<td>0.11%</td>
<td>1.08%</td>
<td>0.11%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>grep</td>
<td>-0.90%</td>
<td>13.06%</td>
<td>-0.21%</td>
<td>1.57%</td>
<td>11.26%</td>
<td>-1.54%</td>
<td>-0.50%</td>
<td>-0.50%</td>
</tr>
<tr>
<td>gzip</td>
<td>0.17%</td>
<td>-0.30%</td>
<td>0.29%</td>
<td>0.39%</td>
<td>9.35%</td>
<td>6.61%</td>
<td>0.00%</td>
<td>-0.17%</td>
</tr>
</tbody>
</table>

TABLE IV. EXPENSE SCORES COMPARISON WITH PEER WORK FOR EIGHT TECHNIQUES

<table>
<thead>
<tr>
<th>Subjects</th>
<th>Tarantula (Revised)</th>
<th>Jaccard (Revised)</th>
<th>Ochiai (Revised)</th>
<th>SBI (Revised)</th>
<th>Wong2 (Revised)</th>
<th>Wong3 (Revised)</th>
<th>O (Revised)</th>
<th>O(^{P}) (Revised)</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>3.85%</td>
<td>3.21%</td>
<td>0.00%</td>
<td>5.31%</td>
<td>29.29%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>0.59%</td>
<td>12.42%</td>
<td>7.77%</td>
<td>12.42%</td>
<td>25.63%</td>
<td>0.39%</td>
<td>0.29%</td>
<td>0.29%</td>
</tr>
<tr>
<td>replace</td>
<td>0.29%</td>
<td>2.93%</td>
<td>0.70%</td>
<td>3.16%</td>
<td>28.74%</td>
<td>0.11%</td>
<td>-0.03%</td>
<td>-0.05%</td>
</tr>
<tr>
<td>schedule</td>
<td>0.67%</td>
<td>4.03%</td>
<td>1.51%</td>
<td>4.53%</td>
<td>9.24%</td>
<td>0.17%</td>
<td>0.17%</td>
<td>0.17%</td>
</tr>
<tr>
<td>schedule2</td>
<td>3.57%</td>
<td>11.82%</td>
<td>7.91%</td>
<td>12.22%</td>
<td>23.50%</td>
<td>0.05%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>tcas</td>
<td>0.22%</td>
<td>3.16%</td>
<td>1.54%</td>
<td>2.68%</td>
<td>19.52%</td>
<td>0.66%</td>
<td>0.09%</td>
<td>0.09%</td>
</tr>
<tr>
<td>tot_info</td>
<td>1.50%</td>
<td>8.84%</td>
<td>5.88%</td>
<td>10.16%</td>
<td>36.10%</td>
<td>0.46%</td>
<td>0.29%</td>
<td>0.29%</td>
</tr>
<tr>
<td>flex</td>
<td>0.33%</td>
<td>0.21%</td>
<td>0.20%</td>
<td>0.11%</td>
<td>1.08%</td>
<td>0.11%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>grep</td>
<td>-0.90%</td>
<td>13.06%</td>
<td>-0.21%</td>
<td>1.57%</td>
<td>11.26%</td>
<td>-1.54%</td>
<td>-0.50%</td>
<td>-0.50%</td>
</tr>
<tr>
<td>gzip</td>
<td>0.17%</td>
<td>-0.30%</td>
<td>0.29%</td>
<td>0.39%</td>
<td>9.35%</td>
<td>6.61%</td>
<td>0.00%</td>
<td>-0.17%</td>
</tr>
</tbody>
</table>

For a more detailed comparison, Table III presents the mean expense reduction scores for eight refined techniques on each program. It can be seen from this table that for most programs and techniques the reduction score is always greater than 0%. This indicates that the effectiveness of conventional SFL techniques get improved by our weighting method in most cases. Among all the improvement, the highest one is in program tot_info using formula Wong2. The relative improvement is as high as 36.10%. Besides, we can also observe that for some technique on certain programs, the relative improvement decrease. For example, the improvement on the program grep using refined Tarantula formula decreases by 0.90%. However, the decrease is very small and limited compared with increase on effectiveness of fault localization. In total, on average the effectiveness of fault localization for 98.6% faulty versions can be enhanced.

E. Results for comparison with peer work

In this section we compare our method with peer works. Naish et al. [5] proposed a weighting function for failed tests and considered weight of a failed test is inversely proportional to the number of statements exercised in the test. We implemented their weighting method on the eight techniques ourselves and carefully examined the
weighting function to assure it is strictly consistent with that in [5]. Table IV shows the mean Expense scores of these techniques on each program. We refer to refined techniques by our method using “-revised” suffix and refined technique by Naish et al. using “-Naish” suffix.

From table IV, we can find that for most of programs, the expense scores enhanced by our method is often greater than the corresponding scores enhanced by Naisht method, which means that when locating faults in programs, our method requires less code examination than the method proposed by Naish. Furthermore, our method can achieve much more improvement on fault localization than method proposed by Naish. Taking Ochiai in Table IV (a) as an example, the expense score is 7.66% using our method in print_tokens2 while 14.35% using Naish method, which indicates that our approach can obtain nearly 50% saving in terms of examination on codes compared with Naish et al. method.

V. CONCLUSIONS AND FUTURE WORKS

In this study, we proposed a weight-based refinement method for conventional SFL techniques to enhance error diagnosis. Our method distinguishes the weight of one failed test case from another or one passed test case from another. For each failed test, different weights are assigned with respect to each statement according to its coverage information and proportion between failed tests and passed tests. For each passed test, weights are assigned according to its average similarity to all failed tests. In such a way, our method can be generally applied to existing SFL techniques to improve fault localization effectiveness. We conduct experiments to evaluate the refinement techniques on eight typical SFL formulas using two standard benchmarks. The experimental results suggested that the revised techniques can always achieve better performance than, or at least competitive with the techniques original performance.

In our future work, we plan to conduct more empirical studies by using larger scale programs and multiple-faults versions. We also wish to explore other factors that may impact on weights of test cases to develop more effective strategies for effectively locating faults.

REFERENCES


[18] http://site.unl.edu/content/sie.php.

