Enhanced Fault Localization by Weighting Test Cases with Multiple Faults

Jaehee Lee 1,2, Jeongho Kim 2, Eunseok Lee 2
1 Software Engineering Laboratory, Software R&D Center, Samsung Electronics, Seoul, South Korea
2 Dept. of Information and Communication Engineering, Sungkyunkwan University, Suwon, South Korea

Abstract - Fault localization is known to be one of the most time-consuming and difficult tasks in the debugging process. Many fault localization techniques have been proposed to automate this step, but they have generally assumed the single-fault situation, which reduces the performance when multiple faults are encountered. This paper proposes new weighting techniques to improve the effectiveness of spectrum-based fault localization by using information extracted from failed test cases that were caused by multiple faults. We evaluate the performance of our technique with six different metrics using the Siemens test suite and space with 159 multiple-fault versions. Based on the experimental study, we observe that our technique outperforms the baselines, in terms of the average expense maximum, by 12%.

Keywords: fault localization, multiple fault, spectrum-based, weighting, classification

1 Introduction

In software engineering, debugging is necessary in order to maintain code quality. However, manually finding the locations of faults is one of the most costly and time-consuming tasks that developers must carry out [1]. In attempts to alleviate this problem, various automatic fault localization techniques have been proposed.

Spectrum-based fault localization (SFL) techniques extract program spectra, which are the execution profiles of program statements and information about whether tests pass or fail [2][3][11][18][22]. This information is used with a ranking metric to rank the program statements according to how likely they are to be buggy.

SFL techniques have been widely used by researchers and developers due to their simplicity, light mechanism, and accuracy. However, they mostly assume only single-fault situations. However, most real world programs contain more than one fault and many failures are caused by multiple faults. The assumption of a single fault degrades the effectiveness of the proposed techniques.

To alleviate this problem, in this paper, we propose a new weighting technique to improve the effectiveness of spectrum-based fault localization by using information extracted from failed test cases that were caused by multiple faults. Our approach is based on the assumption that, if we identify failed test cases executing multiple faults, each statement from these test cases will have a higher likelihood of being buggy than those from test cases executing only a single fault. The empirical results from this study indicate that our approach is promising. The main contributions of this paper are:

1. We propose an enhanced fault localization technique for multiple-fault environments by using weighting to improve the effectiveness of SFL; this is done by including information extracted from failed test cases that were caused by multiple faults.

2. The performance of our technique is verified experimentally. We perform experiments on various types of real programs [16], including the Siemens test suite and space with various metrics containing 159 multiple-fault versions; these are shipped with between two and eight faults.

The remainder of this paper is organized as follows. In Section II, we introduce the necessary background and related work on SFL and the previous research. In Section III, we describe our technique in detail. Section IV goes on to describe the design of our experiments. Section V reports the results and analysis. Threats to the validity of our techniques are discussed in Section VI. Finally, Section VII presents our future work and conclusions.

2 Background and Related work

2.1 Spectrum-based fault localization

Fault localization techniques intend to reduce the cost of debugging by automating the process that is used to find the location of faults in a program. Among them, spectrum-based fault localization (SFL) techniques have been widely used due to their simplicity (i.e., no modeling or complex computation) and relatively high effectiveness [18].

SFL techniques assign a suspicious score to each statement (branches, predicates, and functions can be used) in the program based on the number of passed and failed test cases in the test suite that executed the statement. The basic assumption of SFL is that, if there is failure in a certain
executed test case, a fault exists among the statements that were visited in the test during execution. However, we cannot expect to determine the exact fault location by using only the failed test case. Therefore, the passed test cases are also used to narrow down the fault location.

Table 1 describes some of the notations that are commonly used in the fault localization field. \( h_i \) contains binary information indicating whether the statement was visited or not.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_{11} )</td>
<td>1 1</td>
<td>Statement hit</td>
</tr>
<tr>
<td>( a_{10} )</td>
<td>1 0</td>
<td>Test result</td>
</tr>
<tr>
<td>( a_{01} )</td>
<td>0 1</td>
<td>X</td>
</tr>
<tr>
<td>( a_{00} )</td>
<td>0 0</td>
<td>Pass</td>
</tr>
</tbody>
</table>

\( e_i \) contains binary information that describes the test result (pass or fail). If the test case (\( T_i \)), which is one of the test cases in the test pool, was executed during the runtime and the test result was fail, a certain statement (\( s_i \)) can be described as either \( a_{11} \) (this line was visited) or \( a_{01} \) (this line was not visited). In the same way, if the test result was pass, a certain statement (\( s_i \)) can be described as either \( a_{10} \) (this line was visited) or \( a_{00} \) (this line was not visited). Therefore, according to the test result, every statement will be counted with one of four types of notation (\( a_{11}, a_{10}, a_{01}, a_{00} \)). Table 2 gives an example with five test cases; of these, the first four fail.

<table>
<thead>
<tr>
<th>Table 2 Example of program spectra</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 )</td>
</tr>
<tr>
<td>Stmt1</td>
</tr>
<tr>
<td>Stmt2</td>
</tr>
<tr>
<td>Stmt3</td>
</tr>
<tr>
<td>Stmt4</td>
</tr>
<tr>
<td>Result</td>
</tr>
</tbody>
</table>

A variety of ranking metrics have been proposed and studied, including Tarantula [2], Ochiai [3], Jaccard [21], AMPLE [20], Naish2 [22], GP13 [17], and Hybrid [11]. Each of these calculates suspicious fault ratios in a different way. We describe two representative ranking metrics below.

\[
\text{Tarantula} = \frac{a_{11}}{a_{11} + a_{01} + a_{10}} \\
\]

\[
\text{Naish2} = a_{11} - \frac{a_{10}}{a_{10} + a_{00} + 1} \\
\]

J. A. Jones et al. developed Tarantula [2], which aims to show the suspiciousness of every statement. In addition, they conducted an experimental program based on the language C.

2.2 Effectiveness of SFL in the context of multiple faults

Many approaches have been proposed in an attempt to elucidate fault interactions and their repercussions. J. A. Jones et al. [2] reported that the effectiveness of the techniques declines for all faults as the number of faults increases. Debroy and Wong [4] explored the idea of fault interference by examining the Siemens test suite. N. DiGiuseppe et al. extended their research by classifying fault interactions into one of four types: independent, synergy, obfuscation, and multiple [6]. They also verified the total cost to resolve all faults as the number of faults increases [5].

2.3 Classifying failing test cases

In [8], J. A. Jones and colleagues investigated the use of failure clustering to remove “noise” caused by one fault inhibiting the localization of another. W. Hogerle et al. explored various alternative clustering algorithms to increase parallelism by using algorithms from integer linear programming [10]. However, as they mentioned, relying on the fact that each cluster focuses on a single fault does not seem realistic.

Yu et al. [9] proposed a technique that can be used to distinguish failing test cases that executed a single fault from those that executed multiple faults. To achieve the goal, their technique uses extracted information from a set of fault localization ranked lists, each of which is produced for a certain failing test and the distance between a failing test and the passing test that most resembles it. They mainly aimed to separate failing test cases that executed a single fault in order to apply an existing approach (SFL, automated fault repairing, failure clustering, etc.). Alternatively, our approach focuses on failed test cases executing multiple faults.
2.4 Weighting test cases in SFL

Naish et al. [12] proposed a weighting strategy for failed tests. Failed tests that cover few statements provide more information than other types of failed tests. Thus, they assumed that the weight of a failed test is inversely proportional to the number of statements exercised in the test. In [13], they also proposed an approach where the frequency execution count of each program statement, executed by a respective test, is used.

Bandyopadhyay et al. [14] extended the idea of the nearest neighbor model [18] to utilize the relative importance of different passing test cases for 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.

Y. Li et al. [15] proposed a weight-based refinement for SFL techniques depending on the execution information and the test status. They treated test cases as being unequally important and improved the effectiveness by exploiting varying weights according to the distribution of the test cases.

However, all of these weighting techniques assume a single-fault situation; we expect that their performance will be degraded in a multiple-fault environment.

3 Proposed Approach

In this chapter, we explain the detailed mechanism of our suggested weighting approach. Our approach is based on the assumption that if we identify failed test cases executing multiple faults, each statement of these test cases will be more suspicious than those executing a single fault stochastically. Hence, we assign more weight to statements visited by multiple test cases.

3.1 Overall procedure

In Figure 1, we describe the overall procedure of our proposed approach; this is similar with the general SFL technique. First, the test suite and target subject program are inserted as the input data. Next, the spectrum data are extracted by executing test suites on the subject program. Then, two additional steps are executed. When classifying the test case, failed test cases are classified into one of two groups: single-fault-executing test cases and multiple-fault-executing cases. After classifying the test cases, we grant more weight to multiple-fault-executing test cases. Finally, a ranked list of each statement, in descending order by the suspicious score, is calculated according to each ranking metric.

Any ranking metric can be used with our approach. In this paper, we use Tarantula, Ochiai, Jaccard, AMPLE, Naish2, and GP13, which are popular in the literature.

3.2 Classifying test cases

Basically, we extend Yu’s technique [9] to classify failed test cases as either single-fault-executing test cases or multiple-fault-executing cases. This is done by using the pattern of the spectrum to classify the test cases. We modified their technique by using the hamming distance to calculate the distance between the binary coverage information of the test cases. We checked the accuracy of our results to determine whether this can be used to classify failed test cases; our results were found to be similar to theirs, which indicates that the performance is sufficiently high.

3.3 Weighting test cases

After the multiple-fault-executing test cases were classified, proper weights were granted to each statement that was visited by the test. We used relative weights; for instance, we assigned a weight $\omega_{T1} = 1$ to $a_{11}$ values in the case of single-fault test cases, as is done in the general SFL technique. Alternatively, we assigned a weight $\omega_{T1} = \alpha (\geq 1)$ to $a_{11}$ values in the case of multiple-fault test cases. The $\alpha$ value is introduced as a parameter to represent different weights to multiple test cases. We check the performance of each metric according to variations of the $\alpha$ value.

To calculate $a_{11}$ for each statement, we take the sum of the weights of the failed test cases.

$$a_{11} = \sum_{s,t=1}^{N} \omega_{st} \frac{N}{W}$$

$\omega_{st}$: the relative weight of each failed test case

N: the number of failed test cases

W: the sum of the relative weights $\omega_{st}$

As an example, with data from Table 2, the relative weights for test cases 1-4 are 1.6, 0.8, 0.8, and 0.8 if we assume that T1 is a multiple-fault-executing test case and $\alpha=2$ (N=4.
and $W=5$). Accordingly, the $a_{11}$ values for statements 1-4 are changed to 3.2, 2.4, 3.2, and 1.6, respectively. The $a_{01}$ values can be computed by using a similar weight sum, or we can simply use the total number of failed tests minus $a_{11}$.

In Table 3, we describe the algorithm of our proposed approach:

- TC is the set of all test cases devised for the program being tested;
- Tcn is a subset of TC;
- TP is the set of passed test cases;
- TF is the set of failed test cases;
- TFn is a subset of TF and contains one failed test case;

Table 3 Algorithm of proposed approach

![Algorithm of proposed approach](image)

In order to evaluate and compare the effectiveness of our proposed technique, we investigated the following research questions:

RQ1. How effective is our proposed weighting technique compared to existing (unweighted) approaches?

RQ2. Is there any dependency between the performance of the proposed weighting technique and different subject programs?

RQ3. Is there any dependency between the performance of the proposed weighting technique and different ranking metrics?

RQ4. Do different weighting parameters affect the result? What is the optimal threshold parameter?

4 Experimental Setup

This section presents the experimental setup for the empirical study.

4.1 Subject programs

Our empirical study involved eight subject programs, including the Siemens test suite and space from SIR (Software Infrastructure Repository) [16], along with their faulty versions and test cases. Table 4 provides more information about these subject programs and test cases. Note that, since the test suites for space are relatively large (13525), we opted to randomly select a smaller subset of 738 cases.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Versions</th>
<th>LOC</th>
<th>Test cases</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>7</td>
<td>536</td>
<td>4140</td>
<td>Lexical analyzer</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>10</td>
<td>387</td>
<td>4115</td>
<td>Lexical analyzer</td>
</tr>
<tr>
<td>replace</td>
<td>27</td>
<td>554</td>
<td>5540</td>
<td>Pattern replacement</td>
</tr>
<tr>
<td>schedule</td>
<td>4</td>
<td>425</td>
<td>2650</td>
<td>Priority scheduler</td>
</tr>
<tr>
<td>schedule2</td>
<td>9</td>
<td>766</td>
<td>2710</td>
<td>Priority scheduler</td>
</tr>
<tr>
<td>Tcas</td>
<td>41</td>
<td>173</td>
<td>1578</td>
<td>Altitude separation</td>
</tr>
<tr>
<td>tot_info</td>
<td>23</td>
<td>494</td>
<td>1052</td>
<td>Information measure</td>
</tr>
<tr>
<td>Space</td>
<td>38</td>
<td>6445</td>
<td>13525</td>
<td>Array definition language</td>
</tr>
</tbody>
</table>

4.2 Fault versions

We created 159 multiple-fault versions of the subject programs by taking different combinations of the available faults. We used one-fault version programs, where the fault was in a single line of source code, and discarded versions with runtime errors and those with no failed test cases. In each subject, for each occurrence of multiple faults, we generated up to the number of faulty versions using the SIR. Therefore, the exact number of multiple-fault versions for each subject is the same as the sum of the faulty versions shown in Table 4 (a total of 159 multiple-fault versions).

4.3 Evaluation metrics

According to the fault localization literature [8][18], fault localization evaluation metrics are defined as the percentage of the program that needs to be examined before reaching the first statement (when ranking metrics are used to order executable statements). As Equation (4) indicates, the range of possible values for the fault localization expense varies, and the effectiveness of the employed fault localization technique decreases as the expense value increases. This value is indicative of the time or effort that a developer spends while finding/using the ranks computed by the fault localization technique. This metric, which we refer to as the “expense”, is computed by the following equation:
5 Results and Analysis

In our experiments, we apply the weighted technique to all of the metrics mentioned in Section III and to all of the subject programs listed in Section IV. We also investigate its effectiveness in improving the performance of these SFL techniques on several subject programs. We present our experimental results in Figures 2, 3, 4, and 5. In each figure, the vertical axis (horizontal axis in Figure 2) represents the average expense required to examine the source code. All of the experiments were carried out on a Windows 7 machine with a 3.7 GHz Intel quad-core CPU and 8 GB of memory.

5.1 Effectiveness of the proposed weighting technique

Table 5 shows that our weighting technique achieves better or similar performance than existing unweighted techniques, with an average improvement of 7%. These results are illustrated in Figure 2.

\[
\text{Expense} = \frac{\text{rank of fault}}{\text{size of program}} \times 100 \quad (4)
\]

<table>
<thead>
<tr>
<th>Subject</th>
<th>Unweighted</th>
<th>Weighted</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>2.52</td>
<td>2.27</td>
<td>9.92 %</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>2.69</td>
<td>2.47</td>
<td>8.18 %</td>
</tr>
<tr>
<td>replace</td>
<td>3.40</td>
<td>2.99</td>
<td>12.06 %</td>
</tr>
<tr>
<td>schedule</td>
<td>2.70</td>
<td>2.58</td>
<td>4.44 %</td>
</tr>
<tr>
<td>schedule2</td>
<td>2.71</td>
<td>2.61</td>
<td>3.69 %</td>
</tr>
<tr>
<td>Tcas</td>
<td>3.33</td>
<td>3.07</td>
<td>7.81 %</td>
</tr>
<tr>
<td>tot_info</td>
<td>2.41</td>
<td>2.41</td>
<td>0.00 %</td>
</tr>
<tr>
<td>space</td>
<td>1.39</td>
<td>1.26</td>
<td>9.03 %</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>2.64</td>
<td>2.46</td>
<td>7.02 %</td>
</tr>
</tbody>
</table>

Figure 2 Total average expense (unweighted vs. weighted)

Table 5 Average expense for each subject program

5.2 Different subject programs

An additional consideration is made by looking into each subject program separately. Figure 3 depicts the average expense for each subject program. Our weighting technique outperforms almost all of the subject programs in terms of the average expense (with the exception of a single program). These improvements range between 3.69 % and 12.06 %. Our technique shows equivalent performance to the tot_info program. We speculate this range in performances comes from the characteristics of the program, which affect the results.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Unweighted</th>
<th>Weighted</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tarantula</td>
<td>2.44</td>
<td>2.35</td>
<td>3.69 %</td>
</tr>
<tr>
<td>Ochiai</td>
<td>2.31</td>
<td>2.22</td>
<td>3.90 %</td>
</tr>
<tr>
<td>Jaccard</td>
<td>2.34</td>
<td>2.24</td>
<td>4.27 %</td>
</tr>
<tr>
<td>AMPLE</td>
<td>2.88</td>
<td>2.78</td>
<td>3.47 %</td>
</tr>
<tr>
<td>Naish2</td>
<td>1.84</td>
<td>1.54</td>
<td>16.30 %</td>
</tr>
<tr>
<td>GP13</td>
<td>1.79</td>
<td>1.53</td>
<td>14.53 %</td>
</tr>
</tbody>
</table>

Figure 3 Average expense for each subject program

5.3 Different ranking metrics

Figure 4 visualizes the average expense for each ranking metric. Our weighting technique outperforms all of the metrics in terms of the average expense; this improvement ranges between 3.69 % and 16.30 %. We also speculate that this range in improvement comes from the characteristic of the various ranking metrics. Table 6 shows the results in greater detail.

Table 6 Average expense for each ranking metric

5.4 Weighting parameter effect

To determine the best threshold parameterization, which is the same as finding the optimal weighting value of α, another experiment was conducted. Figure 5 shows the average
expense for different weighting values. This indicates that if the weight is too high the performance can be degraded (as opposed to reducing the expense). We speculate that the optimal $\alpha$ value varies according to characteristics of the subject program and the ranking metric. A study to obtain the optimum value of $\alpha$ for each program and metric will be conducted in the future. In this experiment, to generate preliminary experimental results, we typically fixed the weighting value at $\alpha = 2.25$.

We verified our experiment with six ranking metrics. We expect other suitable fault localization algorithms to deliver similar results; this will also be studied in our future work.

Finally, as mentioned in Section III, we created multiple-fault versions of programs by randomly selecting a number of available faults. It is possible that these faults could be non-representative of real faults. However, due to a lack of fault data for multiple-fault research, many researchers have manipulated multiple-fault versions of programs by using mutation-based fault injection. We made multiple faults by combining existing available faults; this was done because we believe that it is difficult to minimize problems while artificially generating multiple faulty versions in order to simulate realistic faults by the mutant generating process [19].

7 Conclusions And Future Work

In this paper, we proposed an enhanced technique of fault localization for multiple-fault environments. This was done by using weighting to improve the effectiveness of SFL by incorporating information extracted from failed test cases caused by multiple faults. Additionally, we experimentally evaluated the performance of our technique and compared it to various types of real programs and metrics with multiple-fault versions. The results show that our proposed weighting technique can locate faults more precisely than existing unweighted methods. Furthermore, we investigated the dependency between the performance of the proposed weighting technique with different subject programs and ranking metrics.

In the future, we plan to conduct more empirical studies by using large-scale programs, ranking metrics, and multiple-fault versions. Additionally, as mentioned in Section V, we will investigate how the optimal $\alpha$ value changes according to the characteristics of the subject programs and metrics.

8 Acknowledgments

This research was supported by the Next Generation Information Computing Development Program through the National Research Foundation of Korea (NRF), and is funded by the Ministry of Education, Science and Technology (No. 2015045358) and the MISP (Ministry of Science, ICT & Future Planning), Korea, under the National Program for Excellence in Software that is supervised by the IITP (Institute for Information & Communications Technology Promotion).

9 References


[15] Li, Yihan, Chao Liu, and Zi Yuan. "Exploiting Weights of Test Cases to Enhance Fault Localization (S)." SEKE, 2013.


