Improving Weighted-SBFL by Blocking Spectrum

Haruka Yoshioka*, Yoshiki Higo*, Shinji Kusumoto*
*Osaka University, Japan
{h-yosiok, higo, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—Debugging is a costly process in software development, and computer-aided debugging is expected to reduce the cost. In debugging, fault localization is used to identify the location of potentially faulty code. Spectrum-based fault localization (SBFL) identifies program statements that contain faults based on program spectra collected during the execution of the test cases. Conventional SBFL treats all test cases as having equal importance. A weighting technique that assigns importance to test cases based on the similarity of program spectra (where higher similarity indicates higher importance) has been proposed. However, this technique does not significantly improve fault localization accuracy. We attribute this lack of improvement to the presence of sequential program statements, which negatively affect the weighting. In this study, we apply blocking and the weighting of spectra to improve accuracy. We conduct experiments to compare the proposed technique with conventional SBFL and a recent SBFL technique. We show that the proposed technique identifies faulty program statements with higher accuracy than previous SBFL techniques. Weighting based on the similarity of spectra after blocking is thus effective.**

***Index Terms*—Fault Localization, Weighting, Blocking Spectrum, Program Spectrum**

I. Introduction

In software development, debugging is a very costly task. It has been reported that debugging accounts for more than half of software development costs [1], [2]. Computer-assisted debugging is expected to reduce these costs.

In debugging, fault localization is used to identify the location of potentially faulty code. Many fault localization techniques have been proposed [3]–[5].

One of the most actively studied techniques is spectrum-based fault localization (SBFL), which performs fault localization based on program spectra [6]. A program spectrum indicates which parts of a program are executed for a given test case [7]. Compared to other fault localization techniques, SBFL has shown the most promising results [8], [9]. In SBFL, faulty program statements are identified based on the idea that program statements executed in failed test cases are more likely to be faulty and those executed in successful test cases are likely to be less faulty. In conventional SBFL, all test cases are treated as having equal importance. Given a faulty program and test cases as input, SBFL outputs a suspicion value for each statement in the program. The suspicion value (usually in the range of 0 to 1) indicates the likelihood that the statement contains a fault. A higher suspicion value indicates a higher likelihood of a fault.

A weighting technique for SBFL based on the importance of test cases has been proposed [10]. This technique determines the importance based on the similarity of spectra. Successful test cases whose spectra have high similarity to those of failed test cases are given high weights. However, this technique does not significantly improve the accuracy of identifying faulty statements.

We attribute this lack of improvement to the presence of sequentially executed program statements, which negatively affect the weighting. To overcome this problem, we propose the addition of a process called blocking, which combines a sequence of program statements that do not include branches that are executed consecutively into a single block. After blocking, the proposed technique calculates the similarity of blocked spectra and assigns a higher weight to successful test cases that have a higher similarity to failed test cases.

In our experiments, we apply the proposed technique, conventional SBFL, and a recent SBFL technique to a database of faults in open-source software (Defects4J [11]) and compare the ranking of suspicion attached to faulty statements in the top percentile of all program statements. The results confirm that the proposed technique improves the accuracy of identifying faulty program statements compared to those for conventional SBFL and a recent SBFL technique.

The main contributions of this paper are as follows.

- We propose an SBFL technique that weights test cases based on the similarity of spectra with blocking.
- The proposed technique is applied to real faults in open-source software. The results show that it is more effective than existing SBFL techniques.
- We show that blocking effectively improves accuracy.
- We discuss faults for which the proposed technique is ineffective, namely those in conditional predicates.

II. Spectrum-Based Fault Localization

A. Basic Definition

SBFL techniques take a faulty program and its test cases as input. All the given test cases are executed, and then the suspicion value for each program statement is calculated based on the success/failure and spectra for the test cases. The suspicion value is based on the idea that program statements executed in failed test cases are more likely to contain a fault than those executed in successful test cases. The output of SBFL techniques is the suspicion values for the program statements.

Herein, we explain how to calculate the suspicion value for a given program statement. ef^i, nf^i, ep^i, and np^i for s_i are defined as follows:

ef^i number of failed test cases that execute s_i,

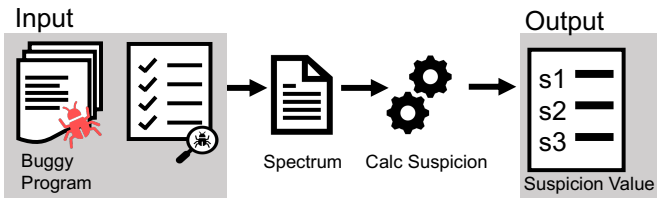

Fig. 1: Flow of SBFL from input to output.

nf^i number of failed test cases that do not execute s_i,
ep^i number of successful test cases that execute s_i, and
np^i number of successful test cases that do not execute s_i.

We obtain a scalar value of suspicion by putting the above values into the formula for calculating suspicion. There are many formulas for calculating suspicion. That proposed by Ochiai [12] is shown below.

$$suspicion(s_i) = \frac{ef^i}{\sqrt{(ef^i + nf^i) * (ep^i + ef^i)}} \quad (1)$$

The rank of a faulty program statement is the index of the fault program statement in a list in which all program statements are sorted in descending order of their suspicion value. If a faulty program statement has the same suspicion value as that of other program statements, the rank of the faulty program statement is set to be lower. For example, if a faulty program statement and another program statement both have the highest suspicion value, the faulty program statement is ranked second.

B. Weighting Test Cases

In conventional SBFL (see Subsection II-A), all test cases are treated equally. A previous study proposed weighting test cases to improve fault localization [10]. Weighting works well for the element tie problem [13]. In this problem, two or more program statements have the same suspicion value [14]. Developers thus do not know which statement to look at first [15], [16]. Assigning unique weights to the test cases would eliminate the element tie problem because the ranking ties would be split.

Studies have proposed SBFL techniques for weighting test cases [10], [17]–[21]. In the present study, we refer to the work of Bandyopadhyay and Ghosh [10], where test cases are weighted based on the similarity of their spectra.

Their techniques, referred to as the BG (Bandyopadhyay and Ghosh) techniques hereafter, assign a weight of below 1 to a successful test case and a weight of 1 to a failed test case.

To compute the weight of a successful test case t_s, the similarity $sim(t_s)$ between t_s and the set of all failed test cases is calculated as shown below, where S_t is the set of program statements executed in test cast t, T^f is the set of all failed test cases, and $|S|$ is the number of elements in set S.

$$sim(t_s, t_f) = \frac{|S_{t_s} \cap S_{t_f}|}{|S_{t_s} \cup S_{t_f}|} \quad (2)$$

$$sim(t_s) = \frac{1}{|T^f|} \sum_{t \in T_f} sim(t_s, f) \quad (3)$$

The similarity $sim(t_s, t_f)$ between a given successful test case t_s and a failed test case t_f is represented by the Jaccard coefficient for the set of program statements executed in t_s and the set of program statements executed in t_f. Then, the similarity $sim(t_s)$ is calculated as the average of the similarities between t_s and each failed test case. By using $sim(t_s)$ and thresholds $thldL$ and $thldH$ ($thldL \leq thldH$), the weight of t_s is calculated as follows.

$$w(t_s) = \begin{cases} 1 - sim(t_s) & (0 \leq sim(t_s) < thld) \\ sim(t_s) & (thldL \leq sim(t_s) \leq thldH) \\ 1 - sim(t_s) & (thldH < sim(t_s) \leq 1) \end{cases} \quad (4)$$

$thldL$ is one of the following three values:

- 0,
- the value of the first quartile in the boxplot of the similarity of all successful test cases, or
- the value of the lower tail in the boxplot.

$thldH$ is one of the following three values:

- 1,
- the value of the third quartile in the boxplot, or
- the value of the upper tail in the boxplot.

Table I shows the codes used for the BG techniques based on the $thldL$ and $thldH$ values. These codes are used in the experiments in Section V.

C. Issues of Prior Work and Key Ideas for Improvement

In the BG techniques, the similarity of the spectra for failed and successful test cases is calculated using the Jaccard coefficient, and the successful test cases are weighted based on the similarity. However, this does not significantly improve the accuracy of fault localization, as shown in experiments. We consider that this lack of improvement is due to the following reasons.

- In the calculation of similarity based on a simple comparison of spectra, consecutively executed program statements that do not contain branches can have an excessive contribution to the similarity of test cases. In other

TABLE I: Codes for the BG techniques for various threshold value combinations

(thldL, thldR)	name
(0, 1)	NoThresh
(lower quartile, 1)	LQ
(lower quartile, upper quartile)	LQ-UQ
(lower tail, 1)	LT
(lower tail, upper tail)	LT-UT
(lower tail, upper quartile)	LT-UQ
(lower quartile, upper tail)	LT-UT

words, inappropriate weights may be assigned because they depend more on the length of consecutively executed statements than on branches. Instead of consecutively executed statements, we consider that branches should be given importance in weighting.
- Successful test cases whose spectra have low similarity to those for failed test cases are also weighted. We consider that successful test cases with low similarity should not be weighted.

We now discuss why branches are important. In a simple comparison of spectra, the similarity of test cases is greatly affected by the existence of program statements that are executed consecutively without branches. In SBFL, the length of consecutive program statements that do not contain branches is not important, but the existence of branches is important. The reason for this is that in SBFL, the coverage information changes only due to branches, and the suspicion value is calculated based on the coverage information. Therefore, we consider that branches should be considered important in the weighting of test cases.

We discuss why weights should not be assigned to successful test cases whose spectra have low similarity to those for failed test cases. We believe that the differences in spectra of failed and successful test cases contain the cause of the success or failure of the test cases. In other words, the fault location is narrowed down by using the differences. A low similarity between a failed test case and a successful one indicates many differences in their spectra, and the fault location is not narrowed down. In such a case, weighting is a noise that reduces the accuracy on the contrary. Therefore, successful test cases with low similarity to failed ones should not be weighted.

III. Proposed Technique

A. Overview

To improve SBFL, the proposed technique assigns weights to test cases based on the idea that a successful test case whose spectra are similar to those for failed test cases is more important. In our technique, the similarity of spectra is obtained after a process called blocking is performed on the target program. Blocking combines a sequence of program statements that do not contain branches into a single block. As described in Subsection II-C, a simple comparison of spectra does not provide a weighting that emphasizes branching. Blocking solves this problem.

Figure 2 shows the procedure for the proposed technique. The proposed technique takes a faulty program and its test cases as input, and outputs a suspicion value for each program statement. The proposed technique consists of the following four steps.

STEP-1: The faulty program is run through the test cases to obtain spectrum information.

STEP-2: Blocks of acquired spectra are made.

STEP-3: The weights for successful test cases are calculated based on the similarity of blocked spectra.

STEP-4: The suspicion value is calculated with the weighting of successful test cases.

The major differences between the BG techniques [10] and the proposed technique are as follows.

- In determining the similarity of spectra, the proposed technique performs blocking, whereas the BG techniques do not.
- In the BG techniques, successful test cases with low similarity are also assigned weights; this is not done in the proposed technique.

Subsection III-B to Subsection III-E describe each step in detail. The proposed technique does not use the Jaccard coefficient for the similarity calculation in STEP-3. The reason for this is given in Subsection III-F. Because our weighting technique is based on blocking, we call the proposed technique Blocked Weighting Spectrum-Based Fault Localization (BWSBFL).

B. ***STEP-1:*** *The faulty program is run through the test cases to obtain spectrum information.*

Spectrum information is obtained using the library JaCoCo [22]. This library is used to obtain a coverage report for a program when passed through the test cases. The spectrum for each test case is obtained by executing each test case one by one and obtaining a coverage report for each execution.

C. ***STEP-2:*** *Blocks of acquired spectra are made.*

Blocking combines statements that are executed consecutively by only certain test cases into a single block. In Figure 3, lines 2-3 are executed consecutively and the set of test cases executing lines 2-3 matches $\{t_a, t_b\}$.Therefore, the blocking process combines lines 2-3 into one block ($B2$). The test cases that execute the seventh and ninth lines in Figure 3 match at t_a, t_b, and t_c. However, the eighth line is executed between the seventh and ninth lines (i.e., the seventh and ninth lines are not performed consecutively), so these lines are grouped into different blocks ($B5$ and $B7$, respectively).

D. ***STEP-3:*** *The weights for successful test cases are calculated based on the similarity of blocked spectra.*

In STEP-3, the similarity between the failed and successful test cases is calculated and the weights of the successful test cases are calculated using this similarity. A threshold $thld$ is set for the similarity. A weight of 1 is assigned when the similarity is below the threshold and a weight of above 1 is assigned when the similarity is greater than the threshold, increasing monotonically as the similarity increases.

In the following, the set of failed test cases is denoted by $\mathbb{F}$. The number of elements in the set $\mathbb{F}$ is denoted by $|\mathbb{F}|$. $only(t_i \to t_j)$ and $both(t_i \to t_j)$ are defined as follows.

$only(t_i \to t_j)$
: Number of program elements executed in test case t_i and not executed in test case t_j.

$both(t_i \to t_j)$
: Number of program elements executed in both test cases t_i and t_j.

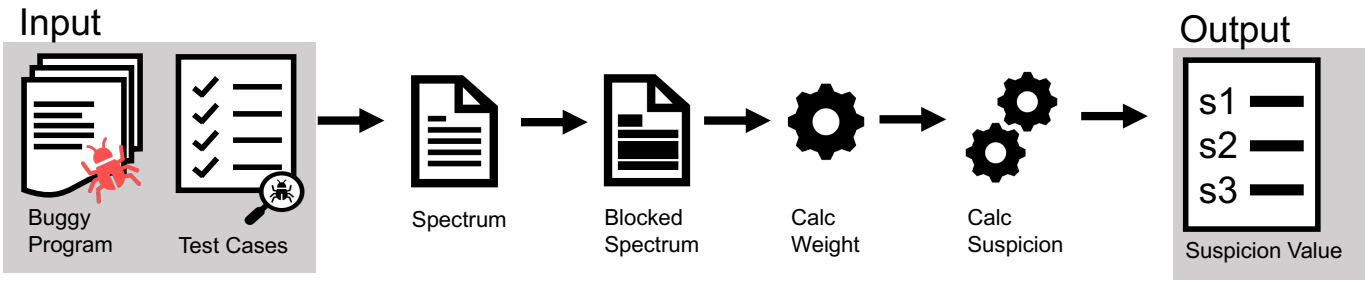

Fig. 2: Flow of proposed technique from input to output.

	int CloseToZero(int n){	ta	tb	tc	Block
1:	if(n>0)	●	●	●	B1
2:	print("n > 0");	●	●		B2
3:	n--;	●	●		
4:	if(n<0)	●	●	●	B3
5:	print("n < 0");			●	B4
6:	n++;			●	
7:	if(n==0)	●	●	●	B5
8:	print("n == 0");	●		●	B6
9:	return n;	●	●	●	B7
	} Result	F	S	S	

● means executed by the test case.
F means the test case failed.
S means the test case successed.

Fig. 3: Example of blocking.

The program elements used in the definition are the blocks described in STEP-1 in BWSBFL.

As an example, we obtain $only(t_a \to t_b)$ and $both(t_a \to t_b)$ in Figure 3. The set of blocks executed in test case t_a is $\{B1, B2, B3, B5, B6, B7\}$ and that executed in test case t_b is $\{B1, B2, B3, B5, B7\}$. Therefore, the set of blocks executed in t_a and not executed in t_b is $\{B6\}$. Therefore, $only(t_a \to t_b) = 1$. The set of blocks executed by both t_a and t_b is $\{B1, B2, B3, B5, B7\}$. Therefore, $both(t_a \to t_b) = 5$.

The similarity of the spectrum of t_j to t_i, $sim\,(t_i \to t_j)$, is defined as follows. We do not use the Jaccard coefficient to compute the similarity between the two sets for the reason given in Subsection III-F.

$$sim\,(t_i \to t_j) = \frac{both(t_i \to t_j)}{only(t_i \to t_j) + both(t_i \to t_j)} \tag{5}$$

The weight $w(t_j)$ for the successful test case t_j is obtained using these values. Let $w(t_i \to t_j)$ be the weight for successful test case t_j relative to failed test case t_i, as defined in equation (6). $w(t_i \to t_j)$ takes 1 when the similarity $sim\,(t_i \to t_j)$ are less than a certain threshold $thld$, and greater than 1 otherwise. The threshold $thld$ is in the range of 0 - 1.

$$w(t_i \to t_j) = \tag{6}$$
$$\begin{cases} 1 & (sim\,(t_i \to t_j) < thld) \\ \dfrac{both(t_i \to t_j)}{\sqrt{only(t_i \to t_j) + both(t_i \to t_j)}} & (thld \leq sim\,(t_i \to t_j)) \end{cases}$$

Because a small weight is assigned when the similarity is lower than the threshold and a large weight is assigned when the similarity is higher, the sigmoid function [23], which is often used in machine learning, can be used as a reference in the weighting. In the range $thld \leq x \leq 1$, the rate of increase of this function becomes smaller as the similarity increases. We added a root to the denominator of the similarity because the weight then takes values similar to the sigmoid function. Note that the function (6) is not the sigmoid function.

The weight $w(t_j)$ for test case t_j is the value obtained by averaging $w(t_i \to t_j)$. It is given as follows.

$$w(t_j) = \frac{\sum_{t_i \in \mathbb{F}} w(t_i \to t_j)}{|\mathbb{F}|} \tag{7}$$

As an example, let us calculate $w(t_c)$ when 0.8 is used as the threshold $thld$ in Figure 3. First, we find $w(t_a \to t_c)$. Because $only(t_a \to t_c) = 1$ and $both(t_a \to t_c) = 5$, the similarity of t_a and t_c is as follows.

$$sim(t_a \to t_c) = \frac{5}{1+5} \fallingdotseq 0.83$$

Because the similarity is above the threshold, $w(t_a \to t_c)$ is $\frac{5}{\sqrt{1+5}} \fallingdotseq 2.04$. Because $\mathbb{F} = \{t_a\}$, $w(c) = \frac{w(t_a \to t_c)}{|\mathbb{F}|} = 2.04$.

E. ***STEP-4:*** *The suspicion value is calculated with the weighting for successful test cases.*

ep'^i and np'^i are defined as follows.

- ep'^i Sum of weights for successful test cases that execute statement S_I.
- np'^i Sum of weights for successful test cases that do not execute statement S_I.

The suspicion value in the proposed technique is calculated by replacing ep^i and np^i in conventional SBFL with ep'^i and np'^i. For example, the formula for calculating the suspicion value in the proposed technique, in which ep^i in Ochiai's formula (1) is replaced by ep'^i, is as follows.

$$suspicion(s_i) = \frac{ef^i}{\sqrt{(ef^i + nf^i) * (ep'^i + ef^i)}} \tag{8}$$

F. Reasons for not Using Jaccard Coefficient for Calculating Similarity

The reasons for not using the Jaccard coefficient, which is often used as a similarity measure, are given below. In the following explanation, t_f is a failed test case and t_s is a successful test case. The Jaccard coefficients for t_f and t_s are expressed as follows using $only(t_i \rightarrow t_j)$ and $both(t_i \rightarrow t_j)$ defined in Subsection III-D.

$$\frac{both(t_f \rightarrow t_s)}{only(t_f \rightarrow t_s) + only(t_s \rightarrow t_f) + both(t_f \rightarrow t_s)} \tag{9}$$

Jaccard coefficients incorporate elements of program statements that are executed only in successful test cases (i.e., $only(t_s \rightarrow t_f)$). The element $only(t_s \rightarrow t_f)$ in equation (9) represents the number of program statements executed in the successful test case t_s and not executed in the failed test case t_f. Because fault localization is based on the idea that the program statements executed in the failed test case contain faults, incorporating features that do not appear in the failed test case into the similarity adds noise. Renieres et al. applied this concept to calculate the similarity between failed and successful test cases [24]. They used the Hamming distance when measuring the similarity between failed and successful test cases. They ignored the fact that the Hamming distance is increased by features that appear in the successful test case but not in the failed test case because these features are undesirable for fault localization. Therefore, in the present study, the elements of the program statements executed only in the successful test case are ignored in calculating the similarity.

IV. Experimental Setup

The purpose of this experiment is to evaluate whether SBFL is more effective in localizing faults when successful test cases are weighted using the similarity of blocked spectra. To evaluate the proposed technique, some existing SBFL techniques are also used. The SBFL techniques used in this experiment are listed below.

- **BG:** The BG techniques [10]. See Subsection II-B for details.
- **Basic:** The basic SBFL technique without weighting [6]. See Subsection II-A for details.
- **BWSBFL:** The proposed technique. Blocking is used to weight successful test cases.
- **WSBFL:** Same as BWSBFL but successful test cases are weighted without blocking.

The differences between the techniques are summarized in Table II. In this experiment, we compare the fault localization accuracy of BWSBFL to those of Basic and BG, evaluate the effect of blocking on fault localization accuracy and the effect of the suspicion value formula on the accuracy of fault localization in BWSBFL, and compare BWSBFL and Basic in terms of execution time. Fault localization techniques are also used for automated program repair (APR). Various APR techniques have been proposed [25]–[27]. Some APR techniques perform fault localization many times before fixing faults [27]. In these techniques, the increase in the execution time for fault localization is related to the increase in the execution time for APR. Therefore, the effect of execution time was examined.

A. Benchmark

In this experiment, we use Defects4J (V1.2.0) [11] as a benchmark. Defects4J is a dataset that includes 395 faults found during the development of six open-source projects written in Java. It is widely used as a benchmark in the study of fault localization [28], [29]. This study uses the Lang, Math, and Time projects included in Defects4J as experimental targets because our tool supports their build tool (Maven). Some of the faults could not be executed, namely ID101-106 (Math), ID41-65 (Lang), and ID22 (Time). The total number of faulty statements in the projects that could be executed was 561 and the number of these statements that were detectable by SBFL was 360. 201 faulty statements were not detectable by SBFL because they are not executed in the failed test case and thus their suspicion value is 0.

B. Formulas for calculating Suspicious Value to be used

To calculate the suspicion value, we use the following formulas: Ochiai [12], DstarN [8], Jaccard [30], Zoltar [31], and Tarantula [32]. These formulas are given below. We set N to 5 for Dstar (11) in this experiment.

Ochiai

$$suspicion(s_i) = \frac{ef^i}{\sqrt{(ef^i + nf^i) * (ep^i + ef^i)}} \tag{10}$$

DstarN

$$suspicion(s_i) = \frac{{ef^i}^N}{nf^i + ep^i} \tag{11}$$

Jaccard

$$suspicion(s_i) = \frac{ef^i}{ef^i + nf^i + ep^i} \tag{12}$$

Zoltar

$$suspicion(s_i) = \frac{ef^i}{ef^i + nf^i + ep^i + \frac{10000 * nf^i * ep^i}{ef^i}} \tag{13}$$

Tarantula

$$suspicion(s_i) = \frac{\frac{ef^i}{ef^i + nf^i}}{\frac{ef^i}{ef^i + nf^i} + \frac{ep^i}{ep^i + np^i}} \tag{14}$$

C. Metric

In this experiment, we use TopN% as an evaluation metric. TopN% indicates the rank of a fault in the top percentile of all statements with suspicion values. Let rank(s) be the rank of a faulty program statement and $|S_{failureExecuted}|$ be the number of statements that the failed test cases executed. Then,

TopN% is expressed as following formula (15). A smaller value indicates higher fault detection accuracy.

$$TopN\% = \frac{rank(s)}{|S_{failureExecuted}|} \tag{15}$$

V. Results and Discussion

The results for Math, Lang, and Time obtained using Ochiai's formula are shown in Figure 4, Figure 5, and Figure 6, respectively. The vertical axis represents the TopN% value and the horizontal axis represents the threshold value in BWSBFL and WSBFL. The solid blue line represents BWSBFL, the solid black line represents WSBFL, the dashed line represents Basic, and the dash-dotted lines represent the BG techniques. NoThresh, LQ, LQ-UQ, LT,... in the legend are the name of the BG techniques shown in Table I. Ochiai's formula is used because Abreu et al. compared several SBFL formulas and concluded that Ochiai's formula is the best [12].

A. Comparison of BWSBFL and Basic

As shown in Figure 4, Figure 5, and Figure 6, Basic tends to show a better TopN% than that for BWSBFL with a low threshold but a worse TopN% than that for BWSBFL with a high threshold. In the proposed technique, a significant weight is given only to successful test cases whose spectra are highly similar to those for failed test cases. Therefore, a high threshold makes BWSBFL outperform Basic.

A comparison of BWSBFL (with a threshold of 0.85) and Basic in terms of detected faults is shown in Table III. The second and third columns show the number of faults for which BWSBFL/Basic was superior to each other. The fourth column indicates the number of faults BWSBFL and Basic gave the same TopN%. For all projects, the number of faults detected by BWSBFL more accurately is higher than that for Basic. In particular, for Lang, the TopN% value for BWSBFL is equal to or greater than that for Basic for all faults. BWSBFL thus outperforms Basic.

B. Comparison of BWSBFL and the BG Techniques

The TopN% results for the various techniques are shown in Figure 4, Figure 5, and Figure 6. The solid blue line represents BWSBFL and the dash-dotted lines represent the BG techniques. BWSBFL with a threshold value of 0.85 shows a better TopN% value than that for the BG techniques for all projects. The BG techniques with some threshold values shows a worse TopN% value than that for Basic. In contrast, BWSBFL with a high threshold value achieves a better TopN% value than that for Basic. For example, BWSBFL with a threshold value of 0.85 is superior to Basic for all projects. These results indicate that BWSBFL has more accurate fault localization than that for the BG techniques.

C. Effect of Blocking on Accuracy of Fault Localization

The TopN% values of BWSBFL and WSBFL for various thresholds are shown in Figure 4, Figure 5, and Figure 6. BWSBFL and WSBFL have the best TopN% values at different thresholds. We use the threshold at which each technique performed best. The threshold at which each technique obtained the best TopN% value and the corresponding TopN% value are shown in Table IV. For Math and Time, BWSBFL has a better TopN% value. For Lang, WSBFL has a better TopN% value.

Table V shows the number of faults for which BWSBFL or WSBFL is superior to each other for the thresholds shown in Table IV. For all projects, BWSBFL has a larger number of correctly detected faults. These results confirm that blocking is effective.

D. Effect of Suspicion Formula on Accuracy of Fault Localization for BWSBFL

The TopN% values for various thresholds were obtained using Jaccard [12], Dstar [8], Zoltar [31], Tarantula [32], and Ochiai [12].

For all projects, the TopN% value decreases when the threshold is around 0.95. For Math and Time, the optimal

TABLE III: Comparison of BWSBFL (threshold: 0.85) and Basic in terms of detected faults

	BWSBFL is superior	Basic is superior	Same
Math	27	21	161
Lang	10	0	83
Time	15	13	30

TABLE IV: Threshold at which BWSBFL and WSBFL show best TopN%

	Threshold		TopN%		
	BWSBFL	WSBFL	BWSBFL	WSBFL	Diff
Math	0.85	0.90	51.22	52.34	1.12
Lang	0.40	0.45	69.35	68.71	-0.64
Time	0.85	0.95	22.01	22.62	0.61

TABLE II: Summary of techniques used in experiment

Name	Blocking	Similarity	Weighting to Test Cases
Basic			always 1
BG		Jaccard	$\begin{cases} 1-sim(t_i) & (0 \le sim(t_i) < thld) \\ sim(t_i) & (thldL \le sim(t_i) \le thldH) \\ 1-sim(t_i) & (thldH < sim(t_i) \le 1) \end{cases}$
BWSBFL	✓	arranged Jaccard	$\begin{cases} 1 & (sim\,(t_i \to t_j) < thld) \\ \dfrac{both(t_i \to t_j)}{\sqrt{only(t_i \to t_j) + both(t_i \to t_j)}} & (thld \le sim\,(t_i \to t_j)) \end{cases}$
WSBFL			

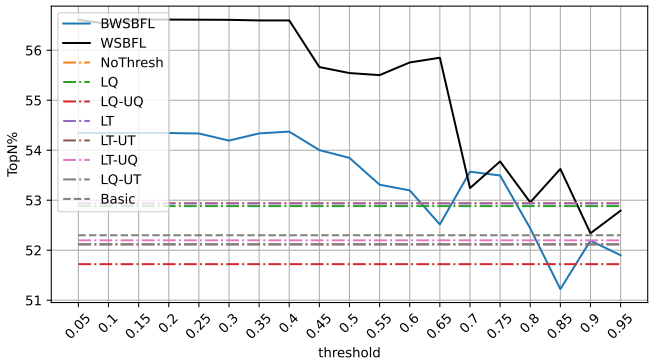

Fig. 4: TopN% comparison results in Math

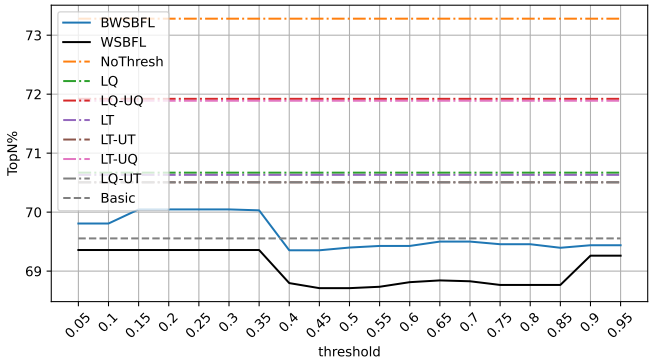

Fig. 5: TopN% comparison results in Lang

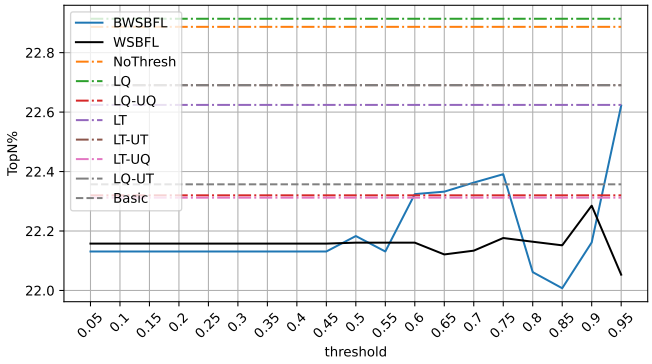

Fig. 6: TopN% comparison results in Time

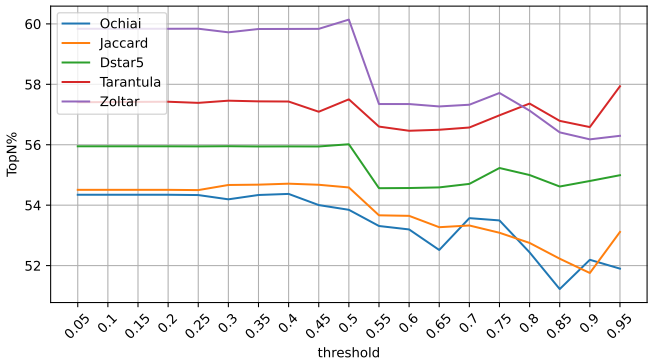

Fig. 7: TopN% obtained using various formulas (Math).

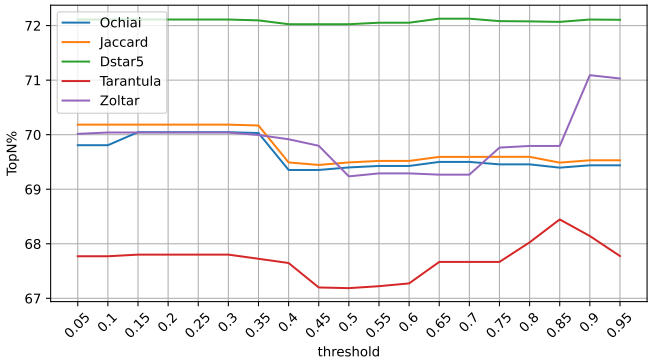

Fig. 8: TopN% obtained using various formulas (Lang).

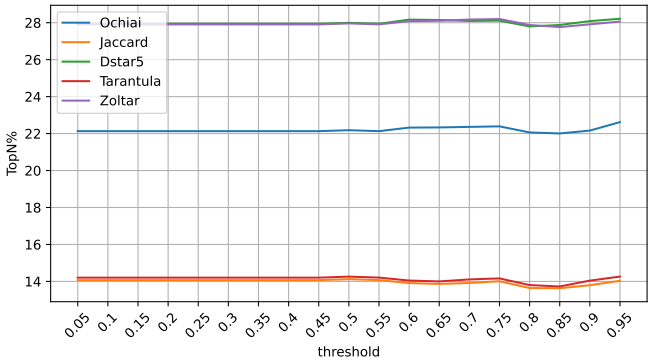

Fig. 9: TopN% obtained using various formulas (Time).

threshold value is 0.85, and for Lang, it is around 0.55. The TopN% value has a similar trend for all calculation formulas.

E. Comparison of BWSBFL and Basic in Terms of Execution Time

Both Basic and BWSBFL have the following two processes.

TABLE V: Number of faults for which BWSBFL and WSBFL outperform each other

	BWSBFL is superior	WSBFL is superior	Same
Math	32	14	163
Lang	4	3	86
Time	10	7	41

1) Obtain spectra by running the program through the test cases.
2) Calculate suspicion value from acquired spectra.

The first process is the same for the two techniques, but the second process is different. Therefore, for comparing the execution time, only the second process is compared. A box-and-whisker diagram of the execution time for each technique is shown in Figure 10.

The average runtime for Basis was 56.4 ms and that for BWSBFL was 67.3 ms. The increase in execution time was thus negligible.

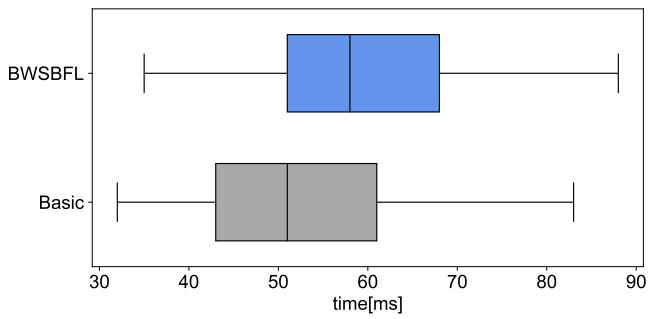

Fig. 10: Execution time for Basic and BWSBFL.

F. Limitations of BWSBFL

We investigated the types of faults for which BWSBFL outperforms Basic. For Defects4J Math, faults that differ in ranking between BWSBFL and Basic are summarized in Table VI. The threshold value for BWSBFL is 0.85. Column 2 indicates the type of statement that contains the fault. Column 3 indicates the modification pattern of the fault. Columns 4 and 5 show the TopN% values for BWSBFL and Basic, respectively. Column 6 shows the difference between the TopN% values of BWSBFL and Basic. A positive (negative) value indicates that BWSBFL (Basic) is superior. The top and bottom parts of the table show the faults for which BWSBFL and Basic are better, respectively. BWSBFL is superior for 27 faults and Basic is superior for 21 faults.

The faults are classified into two types depending on the program statement type: faults in compound statements (While, If, and For Statements) and simple statements.

Of the 27 faults for which BWSBFL is superior, 6 are in compound statements. In contrast, of the 21 faults for which BWSBFL is inferior, 13 are in compound statements. This indicates that BWSBFL tends to not be able to detect faults existing in a compound statement more effectively than Basic.

This is because when there is a fault in the compound statement, the spectrum for the failed test case and that for the successful test case that executes the fault tend to have high similarity. This is clarified using Math 37 as an example.

```
1  public Complex tan(){
2  - if(isNaN){
3  + if(isNaN || Double.isInfinite(real)){
4      return NaN;
5    }
6    ...
7    return createComplex(FastMath.sin(real2)/d,
         FastMath.sinh(imaginary2)/d);
8  }
```

This code shows the location of the fault in Math 37. The second line is the faulty code and the third line is the correct code. When the variable $real$ is $Infinite$, the return statement on line 4 is not executed, and the test case fails. In this case, the spectrum for the failed test case is similar to that for the successful test case when $real$ takes values other than NaN and $Infinite$, so we consider the spectra to be similar.

To support this consideration, we surveyed faults in the Math project that differed in TopN% between BWSBFL and Basic. We compare the percentage of weighted successful test cases among all with the percentage of weighted test cases among the successful test cases that run the fault locations.

There are a total of 557 successful test cases in the Math project in Table VI, of which 167 (30.0%) are weighted successful test cases. There are 557 successful test cases, 100 of which execute faults in the compound statements. Weighted test cases account for 70.0% of these 100 test cases.

This indicates that the test cases that succeed in passing the faults in the compound statement tend to be more similar to the failed test cases. Therefore, the accuracy of BWSBFL for faults in compound statements is lower than that of Basic because the suspicion value for the fault location is lower.

VI. Related Work

Studies on program slicing [33], [34], weighting test cases [17], [18], learning-based fault localization [35]–[37], and fault localization based on the similarity of spectra [24] are summarized below.

A. Program Slicing

Program slicing [33] is used to localize faults. This technique reduces the number of program statements that need to be checked during debugging by eliminating program statements that are irrelevant to the variable of interest. Program slicing can be divided into two main categories. Static slicing [33] analyzes the source code and extracts program statements that may be related to a certain variable. However, static slicing cannot remove program statements that are not actually executed due to branching or other reasons and have no effect on the variable of interest. Dynamic slicing [34] extracts only program statements that affect variables among program statements that are actually executed during program execution. Dynamic slicing takes more time to generate results, but it removes more irrelevant program statements than does static slicing. Program slicing does not assign priorities to program statements in the slice to be examined after irrelevant statements are eliminated. Therefore, it is necessary to go back to the point where the fault is thought to have occurred and examine each program statement one by one and check whether it contains a fault, which may require a lot of effort depending on the program size.

B. Weighting Test Cases

Here, we review some studies that assign weights to test cases. Zhang et al. proposed PRFL, which uses the PageRank algorithm for weighting test cases [17]. The PageRank algorithm determines the importance of a node in a graph based on how it is connected to other nodes. It is based on the idea that nodes that are connected to more nodes and nodes that are connected to nodes of higher importance are more important. In PRFL, each node is a function in the source code. A node has directed edges to the nodes of other functions called within the function. SBFL is performed using the weights assigned

to the test methods as test case weights. PRFL is significantly different from BWSBFL; the former is based on the invocation information between methods, whereas the latter is based on the similarity of spectra. Feature-FL was proposed by Lei et al. [18]. Feature-FL is an SBFL technique that takes into account the probability that each program statement inside conditional predicates is executed. In their technique, program statements that are executed at a higher (lower) probability in failed test cases have higher (lower) suspicion values.

C. Learning-Based Fault Localization

Learning-based fault localization uses machine learning [35]–[37]. This technique uses features such as program invariance and source code complexity in addition to the suspicion value obtained from SBFL to estimate the locations of faults. These features are assigned appropriate weights using a learning-to-rank algorithm [38], a supervised machine learning algorithm often used in search engines.

D. Fault Localization Based on Similarity of Execution Paths

Reiss et al. [24] localized faults based on the similarity of the effective paths for the test cases. In their technique, which is a type of program slicing, the successful test case whose spectrum has the highest similarity to that for the failed test case is first selected. Next, slices of the two test cases (failed and successful) are created and compared. Program statements that only exist in the slice of the failed test case are given to the user as suspicious statements. This technique is not effective

TABLE VI: Type of fault location and TopN% value for Basic and BWSBFL

Subject ID	Statement Type	Modification Patterns	Basic	BWSBFL	Diff
Math 6	Simple Statement (Assignment)	Modify Function Argument	42.01	41.97	0.04
Math 6	Simple Statement (Assignment)	Remove Statement	66.26	65.90	0.36
Math 6	Simple Statement (Function Call)	Modify Increment Statement	66.26	65.90	0.36
Math 6	Simple Statement (Assignment)	Remove Statement	98.35	95.24	3.11
Math 6	Simple Statement (Function Call)	Modify Increment Statement	98.35	95.24	3.11
Math 29	Simple Statement (Assignment)	Remove Statement	44.38	8.28	36.10
Math 29	Compound Statement (While statement)	Modify Conditional Expression	44.38	8.28	36.10
Math 29	Simple Statement (Function Call)	Remove Statement	44.38	8.28	36.10
Math 29	Simple Statement (Function Call)	Modify Function Argument	44.38	8.28	36.10
Math 31	Simple Statement (Variable Declaration)	Remove Statement	52.04	39.33	12.71
Math 31	Simple Statement (Variable Declaration)	Remove Statement	52.04	39.33	12.71
Math 31	Simple Statement (Variable Declaration)	Remove Statement	52.04	39.33	12.71
Math 31	Simple Statement (Variable Declaration)	Modify Assignment	52.04	39.33	12.71
Math 31	Compound Statement (If statement)	Remove Statement	52.04	39.33	12.71
Math 31	Simple Statement (Variable Declaration)	Remove Statement	52.04	39.33	12.71
Math 31	Simple Statement (Assignment)	Remove Statement	52.04	39.33	12.71
Math 31	Simple Statement (Assignment)	Modify Right-hand Code of Assignment	41.73	29.02	12.71
Math 31	Simple Statement (Assignment)	Remove Statement	41.73	29.02	12.71
Math 31	Simple Statement (Assignment)	Remove Statement	41.73	29.02	12.71
Math 31	Simple Statement (Assignment)	Modify Left-hand Code of Assignment	41.73	29.02	12.71
Math 31	Simple Statement (Assignment)	Modify Left-hand Code of Assignment	41.73	29.02	12.71
Math 37	Compound Statement (If Statement)	Modify Conditional Expression	15.52	12.93	2.59
Math 37	Compound Statement (If Statement)	Add If Statement	5.60	3.88	1.72
Math 42	Compound Statement (If Statement)	Add If Statement	26.52	9.68	16.84
Math 64	Simple Statement (Assignment)	Add Assignment Expression	33.99	6.18	27.81
Math 86	Compound Statement (If Statement)	Remove Statement If Statement	52.17	33.15	19.02
Math 98	Simple Statement (Variable Declaration)	Modify Array Index	44.19	23.26	20.93
Math 6	Compound Statement (If Statement)	Modify Conditional Expression	47.22	47.46	-0.24
Math 6	Simple Statement (Increment Statement)	Modify Increment Statement	47.22	47.46	-0.24
Math 6	Simple Statement (Variable Declaration)	Remove Statement	90.44	98.10	-7.66
Math 6	Simple Statement (Function Call)	Modify Increment Statement	90.44	98.10	-7.66
Math 16	Compound Statement (If Statement)	Add Else Branch	16.74	100.00	-83.26
Math 18	Simple Statement (Assignment)	Modify Right-hand Code of Assignment	3.43	4.65	-1.22
Math 18	Simple Statement (Assignment)	Modify Right-hand Code of Assignment	3.43	4.65	-1.22
Math 19	Compound Statement (For Statement)	Add For Statement	11.64	19.50	-7.86
Math 28	Compound Statement (If Statement)	Add If Statement	6.94	15.77	-8.83
Math 33	Compound Statement (If Statement)	Modify Conditional Expression	16.99	17.97	-0.98
Math 37	Compound Statement (If Statement)	Modify Conditional Expression	15.52	41.81	-26.29
Math 37	Compound Statement (If Statement)	Add If Statement	5.60	14.66	-9.06
Math 39	Compound Statement (If Statement)	Add If Statement	76.57	81.64	-5.07
Math 83	Simple Statement (Function Call)	Modify Method Call	27.80	28.29	-0.49
Math 83	Simple Statement (Function Call)	Modify Method Call	27.80	28.29	-0.49
Math 86	Compound Statement (If Statement)	Add If Statement	14.67	16.85	-2.18
Math 88	Simple Statement (Assignment)	Add Assignment Expression	18.59	19.60	-1.01
Math 88	Compound Statement (If Statement)	Add If Statement	18.59	19.60	-1.01
Math 88	Compound Statement (If Statement)	Remove Statement	18.59	19.60	-1.01
Math 88	control statemnet (For Statement)	Remove Statement	18.59	19.60	-1.01
Math 88	Compound Statement (If Statement)	Remove Statement	18.59	19.60	-1.01

Math 6		t_a	t_b	t_c
s1	`int iter = 0; // bug here`	●	●	
s2	`final ConvergenceChecker checker = getConvergenceChecker();`	●	●	
s3	`while(true){`	●	●	
s4	`++iter; // bug here`	●	●	
	`....`	●	●	
	Result	F	S	S

Fig. 11: Defect location for Math 6.

for faults that are located in program statements executed in both failed and successful test cases. In many real-world cases, program statements that contain faults are also executed in successful test cases. An example is ID-6 (Math) in Defects4j. The spectrum information for ID-6 is shown in Figure 11.

$s1$ and $s4$ are faulty program statements, t_a is a failed test case, and t_b and t_c are successful test cases[1]. Test case t_b is the test case whose spectrum is most similar to that for failed test case t_a. Because t_b executes all program statements from $s1$ to $s4$, these faults cannot be found by Reiss et al.'s technique. In addition, the output of their technique is the difference between the failed and successful test cases, which does not provide guidance on where to look first. In contrast, our technique provides a suspicion value that indicates which statements should be examined first.

VII. Threats to Validity

Internal Validity: we implemented the tools, Basic, BWS-BFL, WSBFL, and BG. We take great care to ensure that the tools do not contain flaws. However, there is a possibility that undetected implementation flaws may have affected the results.

External Validity: we used Defects4J (V1.2.0) as a benchmark. Defects4J is a dataset of hundreds of bugs during real-world software development. However, of the six projects in Defects4j (V1.2.0), only three were used as experimental targets. It is also possible that different results would be obtained when the experiment was conducted on the other projects.

VIII. Conclusion

In this study, we proposed a technique that improves the accuracy of SBFL. Our technique uses blocking, and weights successful test cases based on the similarity of their spectra to those for failed test cases. Existing SBFL techniques and the proposed technique were compared in terms of execution time and TopN%.

We confirmed that the TopN% value is improved by the proposed technique at the cost of a slight increase in execution time.

We also investigated the effects of weighting and blocking on accuracy. The results confirm that blocking increases accuracy. The experimental results show that the proposed technique is more effective than existing SBFL techniques.

Our future research includes the followings.

- Our future research will focus on improving accuracy in cases where compound statements contain faults.
- We will optimize the weighting formula (6).
- Using more programs to further enhance the experimental results would be worthwhile.
- We will investigate how blocking and weighting we proposed affects other state-of-the-art SBFL techniques, such as PRFL [17], Feature-FL [18], or learning-based fault localization [35]–[37].

[1]Some parts of the source code have been changed due to space limitations and some test cases are omitted.

Acknowledgments

This research was supported by JSPS KAKENHI Japan (JP20H04166, JP21K18302, JP21K11820, JP21H04877, JP22H03567, JP22K11985)

References

[1] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002.

[2] T. Britton, L. Jeng, G. Carver, and P. Cheak, "Quantify the time and cost saved using reversible debuggers," Cambridge Judge Business School, Tech. Rep., 2012.

[3] B. Korel, "Pelas-program error-locating assistant system," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1253–1260, 1988.

[4] W. Jin and A. Orso, "F3: Fault localization for field failures," in *Proc. International Symposium on Software Testing and Analysis*, 2013, pp. 213–223.

[5] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: Statistical model-based bug localization," in *Proc. Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005, pp. 286–295.

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

[7] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

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

[9] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proc. IEEE/ACM 39th International Conference on Software Engineering*, 2017, pp. 609–620.

[10] A. Bandyopadhyay and S. Ghosh, "Proximity based weighting of test cases to improve spectrum based fault localization," in *Proc. International Conference on Automated Software Engineering*, 2011, pp. 420–423.

[11] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[12] R. Abreu, P. Zoeteweij, and A. J. van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, 2007, pp. 89–98.

[13] V. Debroy, W. E. Wong, X. Xu, and B. Choi, "A grouping-based strategy to improve the effectiveness of fault localization techniques," in *Proc. 10th International Conference on Quality Software*, 2010, pp. 13–22.

[14] M. Golagha and A. Pretschner, "Challenges of operationalizing spectrum-based fault localization from a data-centric perspective," in *Proc. IEEE International Conference on Software Testing, Verification and Validation Workshops*, 2017, pp. 379–381.

[15] X. Xu, V. Debroy, W. E. Wong, and D. Guo, "Ties within fault localization rankings: Exposing and addressing the problem," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 21, pp. 803–827, 2011.

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

[17] M. Zhang, Y. Li, X. Li, L. Chen, Y. Zhang, L. Zhang, and S. Khurshid, "An empirical study of boosting spectrum-based fault localization via pagerank," *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1089–1113, 2021.

[18] Y. Lei, H. Xie, T. Zhang, M. Yan, Z. Xu, and C. Sun, "Feature-fl: Feature-based fault localization," *IEEE Transactions on Reliability*, vol. 71, no. 1, pp. 264–283, 2022.

[19] B. Jiang and W. Chan, "On the integration of test adequacy, test case prioritization, and statistical fault localization," in *Proc. 10th International Conference on Quality Software*, 2010, pp. 377–384.

[20] T. Dao, M. Wang, and N. Meng, "Exploring the triggering modes of spectrum-based fault localization: An industrial case," in *Proc. 14th IEEE Conference on Software Testing, Verification and Validation*, 2021, pp. 406–416.

[21] C. Gong, Z. Zheng, W. Li, and P. Hao, "Effects of class imbalance in test suites: An empirical study of spectrum-based fault localization," in *Proc. IEEE 36th Annual Computer Software and Applications Conference Workshops*, 2012, pp. 470–475.

[22] H. M.R., "Jacoco java code coverage library." https://www.eclemma.org/jacoco/, [Online; accessed 12-December-2021].

[23] A. Menon, K. Mehrotra, C. K. Mohan, and S. Ranka, "Characterization of a class of sigmoid functions with applications to neural networks," *Neural Networks*, vol. 9, no. 5, pp. 819–835, 1996.

[24] M. Renieres and S. Reiss, "Fault localization with nearest neighbor queries," in *Proc. 18th IEEE International Conference on Automated Software Engineering*, 2003, pp. 30–39.

[25] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

[26] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, no. 1, 2018.

[27] Y. Higo, S. Matsumoto, R. Arima, A. Tanikado, K. Naitou, J. Matsumoto, Y. Tomida, and S. Kusumoto, "kGenProg: A high-performance, high-extensibility and high-portability apr system," in *Proc. Asia-Pacific Software Engineering Conference*, 2018, pp. 697–698.

[28] A. Ghanbari, S. Benton, and L. Zhang, *Practical Program Repair via Bytecode Mutation*, 2019, p. 19–30.

[29] X. Li, S. Zhu, M. d' Amorim, and A. Orso, "Enlightened debugging," in *Proc. IEEE/ACM 40th International Conference on Software Engineering*, 2018, pp. 82–92.

[30] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: problem determination in large, dynamic internet services," in *Proc. International Conference on Dependable Systems and Networks*, 2002, pp. 595–604.

[31] S. Heiden, L. Grunske, T. Kehrer, F. Keller, A. van Hoorn, A. Filieri, and D. Lo, "An evaluation of pure spectrum-based fault localization techniques for large-scale software systems," *Software: Practice and Experience*, vol. 49, pp. 1197–1224, 2019.

[32] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proc. International Conference on Automated Software Engineering*, 2005, pp. 273–282.

[33] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984.

[34] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *SIGPLAN Not.*, vol. 25, no. 6, p. 246–256, 1990.

[35] J. Xuan and M. Monperrus, "Learning to Combine Multiple Ranking Metrics for Fault Localization," in *Proc. 30th International Conference on Software Maintenance and Evolution*, 2014.

[36] T.-D. Le, D. Lo, C. Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proc. 25th International Symposium on Software Testing and Analysis*, 2016, pp. 177–188.

[37] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, 2017.

[38] T.-Y. Liu, "Learning to rank for information retrieval," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 3, pp. 225–331, 2009.