Enhancing Spectrum Based Fault Localization Via Emphasizing Its Formulas With Importance Weight

Qusay Idrees Sarhan
Department of Software Engineering, University of Szeged
Szeged, Hungary
Department of Computer Science, University of Duhok
Duhok, Iraq
sarhan@inf.u-szeged.hu

ABSTRACT

Spectrum-Based Fault Localization (SBFL) computes suspicion scores, using risk evaluation formulas, for program elements (e.g., statements, methods, or classes) by counting how often each element is executed or not executed by passing versus failing test cases. The elements are then ranked from most suspicious to least suspicious based on their scores. The elements with the highest scores are thought to be the most faulty. The final ranking list of program elements helps testers during the debugging process when attempting to locate the source of a bug in the program under test. In this paper, we present an approach that gives more importance to program elements that are executed by more failed test cases compared to other elements. In essence, we are emphasizing the failing test cases factor because there are comparably much less failing tests than passing ones. We multiply each element's suspicion score obtained by an SBFL formula by this importance weight, which is the ratio of covering failing tests over all failing tests. The proposed approach can be applied to SBFL formulas without modifying their structures. The experimental results of our study show that our approach achieved a better performance in terms of average ranking compared to the underlying SBFL formulas. It also improved the Top-N categories and increased the number of cases in which the faulty method became the top-ranked element.

KEYWORDS

Debugging, fault localization, spectrum-based fault localization, importance weight, suspiciousness score.

ACM Reference Format:
Qusay Idrees Sarhan. 2022. Enhancing Spectrum Based Fault Localization Via Emphasizing Its Formulas With Importance Weight. In *International Workshop on Automated Program Repair (APR'22), May 19, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3524459.3527349

ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9285-3/22/05…$15.00
https://doi.org/10.1145/3524459.3527349

1 INTRODUCTION

Many aspects of our daily lives are automated by software. They are, however, far from being faultless. Software bugs can result in dangerous situations, including death. As a result, various software fault localization techniques, such as spectrum-based fault localization (SBFL) [Wong et al. 2016], have been proposed over the last few decades. SBFL calculates the likelihood of each program element of being faulty based on program spectra collected from executing test cases and their results. SBFL, on the other hand, is not yet widely used in the industry due to a number of challenges and issues [Sarhan and Beszédes 2022].

One of such issues is that program elements are ranked from most to least suspicious in order of their suspicion scores. Testers check each element starting at the top of the ranking list to determine whether it is faulty or not. Thus, the faulty element should be placed near the top of the ranking list to aid testers in discovering it early and with the least effort. Many times, the underlying SBFL formulas place the faulty elements far from the ranking list top.

In this paper, we are addressing this issue by presenting an approach that gives more importance to program elements that are executed by more failed test cases compared to other elements. The intuition is the following. A typical SBFL matrix is unbalanced in the sense that there are much more passing tests than failing ones, and many SBFL formulas treat passing and failing tests similarly. We propose to emphasize the factor of the failing tests in the formulas, which is achieved by introducing a multiplication factor to SBFL formulas. This factor is called *the importance weight* and is given as the ratio of executed failing tests for a program element with respect to all failing tests. In other words, a program element will be more suspicious if it is affected by a larger portion of the failing tests. The proposed approach can be applied to SBFL formulas without modifying their structures.

The experimental results of our study show that our approach achieved a better performance in terms of average ranking compared to the underlying SBFL formulas. Buggy elements' rankings improved by an average of 3 positions. Also, it achieved positive improvements in all the Top-N categories, and in particular, the number of cases where the faulty element moved to the top of the ranking list increased by 4–23%.

The following are the paper's main contributions:

(1) A new approach that successfully improves the performance of SBFL in many cases is proposed.
(2) The impact of the new approach on the overall effectiveness of SBFL is discussed.

And, our concrete Research Questions (RQs) are:

- **RQ1:** What level of average ranks improvements can we achieve using the proposed approach?
- **RQ2:** What is the impact of the proposed approach on SBFL effectiveness across the Top-N categories?

2 BACKGROUND OF SBFL

Many techniques have been proposed in the literature to automate software fault localization [Wong et al. 2016]. However, SBFL is the most dominant because of its straightforward but potent nature, i.e. it only uses test coverage and results to calculate suspiciousness scores for program elements. Thus, this section explains SBFL and how it can be used to find software faults by ranking program elements according to their likelihood of being faulty.

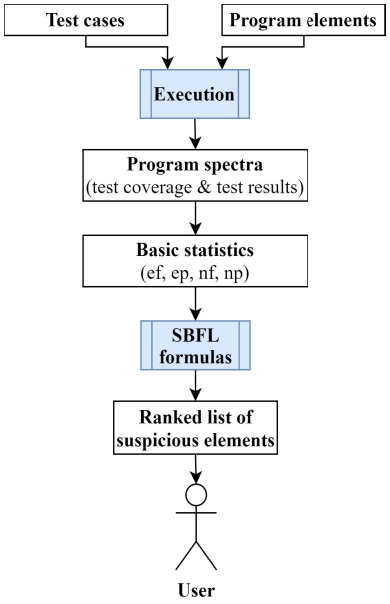

Figure 1: SBFL process

Figure 1 shows the SBFL process. The execution of test cases on program elements is recorded to extract the spectra (i.e., tests coverage and test results) for the program under test. Program spectra information is represented as a matrix. The tests are represented by the columns, while the program elements are represented by the rows. If a test case covers an element of the matrix, it becomes 1; otherwise, it becomes 0. The test results are also stored in the matrix (i.e., the last row), where 0 denotes a passing test case and 1 denotes a failing test case.

Using program spectra and for each program element e, the following four basic statistical numbers are computed:

- **ep**: represents the number of passed test cases covering the program element e.
- **ef**: represents the number of failed test cases covering the program element e.
- **np**: represents the number of passed test cases not covering the program element e.
- **nf**: represents the number of failed test cases not covering the program element e.

Then, these four basic statistics can be used by an SBFL formula to suggest a ranked list of suspicious elements as an output for the tester. The element with the highest ranking on the list is the most likely to have a fault. As a result, SBFL can make it easier for developers to locate the faults in the target program's code.

To illustrate the work of SBFL, consider a Python program that performs some specific mathematical operations, comprises four methods M_i $(1 \leq i \leq 4)$ and six test cases T_j $(1 \leq j \leq 6)$, as shown in Figure 2. The program has a fault in the method M1 (the first statement should be z = x + y). The test cases have been executed on the program and then the execution information (also called spectra) of the four methods in passed and failed test cases have been recorded as presented in Table 1. The program spectra is then used by an SBFL formula such as Tarantula, see Table 6, to compute the suspiciousness of each method of being faulty. Table 2 presents the scores and ranks of the methods of our Python code example. It can be noted that the method M1 is ranked 1 while the others share the rank 3; thus, the method M1 should be examined before the others.

```
M1: def add(x, y):
        z = x * y
        return z

M2: def sub(x, y):
        z = x - y
        return z

M3: def mult(x, y):
        z = x * y
        return z

M4: def div(x, y):
        z = x / y
        return z
```

```
import math_functions

def test_T1():
    assert math_functions.add(3, 3) == 6

def test_T2():
    assert math_functions.add(2, 2) == 4

def test_T3():
    assert math_functions.sub(3, 2) == 1

def test_T4():
    assert math_functions.mult(5, 5) == 25

def test_T5():
    assert math_functions.div(4, 2) == 2

def test_T6():
    assert math_functions.div(8, 2) == 4
```

Figure 2: Running example – code and test cases

Table 1: Running example – spectra and basic statistics

	T1	T2	T3	T4	T5	T6	ef	ep	nf	np
M1	1	1	0	0	0	0	1	1	0	4
M2	0	0	1	0	0	0	0	1	1	4
M3	0	0	0	1	0	0	0	1	1	4
M4	0	0	0	0	1	1	0	2	1	3
Results	1	0	0	0	0	0				

Table 2: Running example – scores and ranks

	Tarantula score	Rank
M1	0.83	1
M2	0.0	3
M3	0.0	3
M4	0.0	3

3 RELATED WORKS

This section briefly presents the most relevant works of improving SBFL by targeting its formulas.

Forming new SBFL formulas is one of the ways of improving SBFL. Here, the researchers attempt to introduce new SBFL formulas that outperform the existing ones. For example, the authors in [Wong et al. 2014] proposed a new SBFL formula called "DStar". The proposed formula has been compared with several widely used formulas and it showed good performance compared to others. SBFL formulas can also be automatically generated by using Genetic Programming (GP). The authors in [Ajibode et al. 2020] used GP to automatically design SBFL formulas directly from the program spectra. The authors were able to produce 30 different formulas. Their results concluded that the GP is a good approach to produce effective formulas for SBFL.

Modifying existing SBFL formulas also leads to improvements. The authors in [You et al. 2013] also modified three well-known SBFL formulas based on the idea that some failed tests may provide more information than other failed ones. Therefore, for the three used formulas, different weights for failed tests were assigned and then applied with multi-coverage spectra.

A different approach is to combine existing SBFL formulas with each other. The authors in [Bagheri et al. 2019] proposed a method for generating a new SBFL formula tailored to a certain program by combining 40 different formulas. The proposed method extracts information from the program using mutation testing and then combines multiple formulas based on the gathered information using different voting systems to generate a new formula. The results of the experiments show that the formula generated by their method is better than several existing ones. It is worth mentioning that researchers tried to merge numerous formulas to create new ones. Because the benefits of several existing formulas have been merged, the new formula is known as a hybrid formula. The performance of a hybrid formula should be superior to that of existing formulas, as shown in [Kim et al. 2015; Park et al. 2014].

Another interesting way is to add new information to existing SBFL formulas. The authors in [Vancsics et al. 2021] utilized the method calls frequency of the subject programs during the execution of failed tests to add new contextual information to the standard SBFL formulas. Here, the frequency ef was substituted for the ef in each formula. The results of their study demonstrated that employing new information from method calls into the underlying formulas can improve SBFL effectiveness.

All the studies mentioned in this section improved the performance of SBFL formulas in different ways. Our proposed approach improves the SBFL performance by giving more importance to program elements that are executed by more failed test cases. The advantages of our proposed approach over others are (a) it does not modify the existing SBFL formulas. Thus, it could be applied to a wide range of SBFL formulas to enhance their effectiveness. This is very important as it makes the proposed approach more applicable than other approaches. (b) it does not require any additional information from a program and the execution of its tests rather than the basic statistics (i.e., ef and nf) calculated from program spectra. Thus, it is efficient and generalizable. (c) it solves the issue of an unbalanced SBFL matrix in the sense that there are much more passing tests than failing ones, and many SBFL formulas treat passing and failing tests similarly.

4 THE PROPOSED SBFL ENHANCING APPROACH

In this section, we present the concept of our proposed approach to enhance the effectiveness of the underlying SBFL formulas and how it works. Then, we present its effectiveness when applied to our motivational example.

4.1 The proposed approach

Figure 3 shows our proposed approach. Using the selected SBFL formulas on the program spectra, we calculate the suspicion scores of program methods. The output is the initial suspicion scores of methods. Then, we multiply each initial score of each method by its importance weight which is computed via $ef/(ef + nf)$ of each corresponding method. This will improve the initial ranking list by emphasizing the methods that are executed by more failing tests and lowering the rank of the methods that are executed by less failing tests. As a result, a final improved ranking list is produced.

4.2 A motivation example from Defects4J

To show how our proposed approach works and how it achieves improvements, several bugs from the used Defects4J dataset were carefully examined. The bug 6 from the "Chart" project was one of the more interesting cases we looked into[1]. Thus, we will illustrate on the basic statistics extracted from the spectra of 26 methods (M1-M26), including the faulty method M21, as presented in Table 3.

Tarantula formula was applied on the extracted execution information to compute the suspicion score of each method as presented in Table 4. It can be seen that the underlying Tarantula formula cannot put the faulty method M21, it is ranked 13 based on Equation 1, near the top of the ranking list suggested by the formula. The reason is that Tarantula assigned higher scores to other 11 methods (i.e., M6, M7, M15-M17, M19, and M22-M26) that have been executed by less number of failed test cases (i.e, one failed test). As a result, these methods got higher ranks in the ranking list and will be examined before the actual faulty method M21.

In our example, the faulty method M21 was executed by two failed test cases. As the method M21 was executed by more failed test cases compared to the other 11 methods, it should be the most suspicious method and it should get more importance than the other 11 methods. To achieve this, we multiply the initial suspicion score of each method by the *importance weight* $ef/(ef + nf)$ of each one.

[1] http://program-repair.org/defects4j-dissection/#!/bug/Chart/6

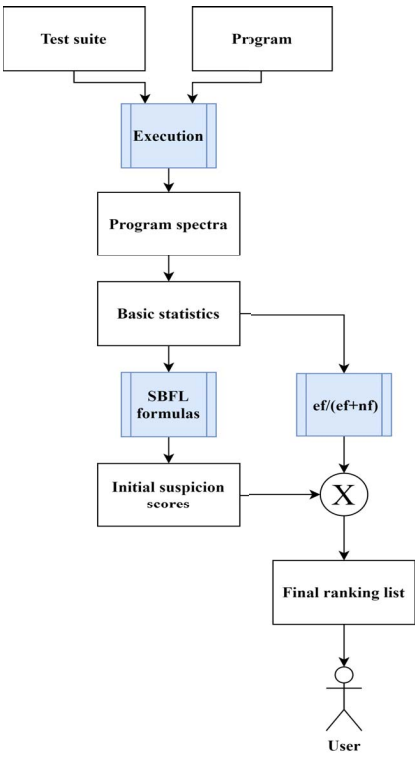

Figure 3: The proposed approach

Table 3: Motivation example's basic statistics

		ef	ep	nf	np
M1	..chart.util.SerialUtilities.readShape()	1	121	1	1759
M2	..chart.util.SerialUtilities.writeShape()	1	121	1	1759
M3	..chart.util.SerialUtilities.class$()	1	147	1	1733
M4	..chart.util.ObjectUtilities.<clinit>()	1	221	1	1659
M5	..chart.util.ObjectUtilities.equal()	2	683	0	1197
M6	..chart.util.HashUtilities.hashCode(I)	1	47	1	1833
M7	..chart.util.HashUtilities.hashCode(II)	1	54	1	1826
M8	..chart.util.AbstractObjectList.<init>()	2	522	0	1358
M9	..chart.util.AbstractObjectList.<init>(I)	2	522	0	1358
M10	..chart.util.AbstractObjectList.<init>(II)	2	522	0	1358
M11	..chart.util.AbstractObjectList.get()	2	237	0	1643
M12	..chart.util.AbstractObjectList.set()	2	240	0	1640
M13	..chart.util.AbstractObjectList.size()	2	429	0	1451
M14	..chart.util.AbstractObjectList.equals()	2	259	0	1621
M15	..chart.util.AbstractObjectList.hashCode()	1	47	1	1833
M16	..chart.util.AbstractObjectList.writeObject()	1	78	1	1802
M17	..chart.util.AbstractObjectList.readObject()	1	78	1	1802
M18	..chart.util.ShapeList.<init>()	2	429	0	1451
M19	..chart.util.ShapeList.getShape()	1	21	1	1859
M20	..chart.util.ShapeList.setShape()	2	25	0	1855
M21	**..chart.util.ShapeList.equals()**	**2**	**221**	**0**	**1659**
M22	..chart.util.ShapeList.hashCode()	1	0	1	1880
M23	..chart.util.ShapeList.writeObject()	1	65	1	1815
M24	..chart.util.ShapeList.readObject()	1	65	1	1815
M25	..chart.util.junit.ShapeListTests.testEquals()	1	0	1	1880
M26	..chart.util.junit.ShapeListTests.testSerialization()	1	0	1	1880

This will emphasize methods that are executed by more failing tests and lower the rank of methods that are executed by less failing tests. The initial Tarantula suspicion score for the faulty method M21 is 0.895 and its importance weight will be 2/(2+0)=1.0. Thus, the final score of M21 will be the same which is 0.895x1.0=0.895. However, the initial scores of all the other 11 methods will be reduced and the ranks will be lower. For example, the final score of the method M6 will be 0.952x0.5=0.476 and its final rank will be 16.5 instead of 6.5. It can be noted that the faulty method M21 has the second most suspicion score and thus ranked the nearest top in the ranking list after applying our proposed approach.

Table 4: Motivation example – scores and ranks

	T score	T rank	T* score	T* rank
M1	0.886	16.5	0.443	23.5
M2	0.886	16.5	0.443	23.5
M3	0.865	19	0.432	25
M4	0.810	22	0.405	26
M5	0.734	26	0.734	11
M6	0.952	6.5	0.476	16.5
M7	0.946	8	0.473	18
M8	0.783	24	0.783	9
M9	0.783	24	0.783	9
M10	0.783	24	0.783	9
M11	0.888	14	0.888	3
M12	0.887	15	0.887	4
M13	0.814	20.5	0.814	6.5
M14	0.879	18	0.879	5
M15	0.952	6.5	0.476	16.5
M16	0.923	11.5	0.462	21.5
M17	0.923	11.5	0.462	21.5
M18	0.814	20.5	0.814	6.5
M19	0.978	5	0.489	15
M20	0.987	4	0.987	1
M21	**0.895**	**13**	**0.895**	**2**
M22	1.000	2	0.500	13
M23	0.935	9.5	0.468	19.5
M24	0.935	9.5	0.468	19.5
M25	1.000	2	0.500	13
M26	1.000	2	0.500	13

5 EVALUATION

5.1 Subject programs

In this study, we used the single faults programs of Defects4J v1.5.0, a common benchmark dataset used in fault localization research [Just et al. 2014; Li and Zhang 2017], where 6 open-source Java programs have 302 real single faults that were identified and extracted from the projects' repositories[2]. However, due to instrumentation issues, 5 faults were not used in this study. Therefore, 297 faults were used in our final dataset. Each program's primary characteristics are presented in Table 5.

[2]https://github.com/rjust/defects4j/tree/v1.5.0

Table 6: SBFL formulas

	Formulas
Tarantula (T)	$\frac{\frac{ef}{ef+nf}}{\frac{ef}{ef+nf}+\frac{ep}{ep+np}}$
Ochiai (O)	$\frac{ef}{\sqrt{(ef+nf)*(ef+ep)}}$
Jaccard (J)	$\frac{ef}{ef+nf+ep}$
Barinel (B)	$\frac{ef}{ef+ep}$
SorensenDice (S)	$\frac{2*ef}{2*ef+nf+ep}$
DStar (DS)	$\frac{ef*ef}{ep+nf}$
Dice (D)	$\frac{2*ef}{ef+nf+ep}$
Interest (I)	$\frac{ef}{(ef+nf)*(ef+ep)}$
Baroni (BR)	$\frac{\sqrt{ef*np+ef}}{\sqrt{ef*np+ef+nf+ep}}$
Kulczynski1 (K)	$\frac{ef}{nf+ep}$
Cohen (C)	$\frac{2*(ef*np)-2*(nf*ep)}{(ef+ep)*(ep+np)+(nf+np)*(ef+nf)}$

Table 5: Subject programs

Project	Number of bugs	Size (KLOC)	Number of tests	Number of methods
Chart	16	96	2.2k	5.2k
Closure	113	91	7.9k	8.4k
Lang	47	22	2.3k	2.4k
Math	77	84	4.4k	6.4k
Mockito	25	11	1.3k	1.4k
Time	19	28	4.0k	3.6k
All	297	332	22.1k	27.4k

5.2 Granularity of data collection

Method-level granularity was used as a program spectra/coverage type in this work. It provides users with a more understandable level of abstraction [B. Le et al. 2016; Zou et al. 2021]. However, in terms of the proposed approach, there is no barrier to investigate other granularity levels too.

5.3 Evaluation baselines

In this paper, 11 widely-studied SBFL formulas [Abreu et al. 2006; Abreu et al. 2007; Abreu et al. 2009; Jones et al. 2002; Naish et al. 2011; Sørensen 1948; Wong et al. 2014], presented in Table 6, were used as benchmarks against our proposed approach. It is worth mentioning that the approaches mentioned in Section 3 are not directly comparable to ours as they applied modifications to the SBFL formulas. Our proposed approach can be applied to an SBFL formula without modifying the formula itself.

6 EXPERIMENTAL RESULTS AND DISCUSSION

This section presents and discusses the overall impact of the proposed approach on SBFL effectiveness. We use evaluation metrics that have been used also by other researchers in the literature for this purpose [Beszédes et al. 2020; Jiang et al. 2019; Xu et al. 2011].

6.1 Achieved improvements in average ranks

We use the average rank approach in Equation 1, where S denotes the tie's starting position and E denotes the tie's size, to analyze SBFL efficiency in general. Here, the program elements with the same suspicion score are ranked using the average rank, such elements are called *tied elements* and they are prevalent in software fault localization [Sarhan et al. 2021], by averaging their positions after they have been sorted in descending order according to their scores.

$$\text{MID} = S + \left(\frac{\text{E - 1}}{2}\right) \tag{1}$$

Table 7 presents the average ranks before (column 2) and after (column 3) applying our proposed approach and also the difference between the average ranks (column 4) in both cases. If the difference is negative, it indicates that our proposed approach has the potential to improve.

Table 7: Comparison of average ranks

	Before	After	Diff.
Tarantula	83.05	76.94	-6.11
Ochiai	79.25	77.99	-1.26
Jaccard	82.08	79.05	-3.03
Barinel	83.05	79.24	-3.81
SorensenDice	82.08	79.01	-3.07
DStar	238.01	237.34	-0.67
Dice	82.08	79.05	-3.03
Interest	83.05	79.24	-3.81
Baroni	85.21	80.62	-4.59
Kulczynski1	240.98	238.01	-2.97
Cohen	86.56	79.84	-6.72

With all of the selected SBFL formulas, we can observe that our proposed approach improved the average rank; reduced by more than 3 overall, which corresponds to 0.26–2.26% of the total number of methods in the used dataset. It can be noted that the formulas Cohen and Tarantula reduced the average ranks quite a lot compared to other formulas. Considering the formulas that have lower average ranks after applying our proposed approach, Tarantula and Ochiai are the best ones, respectively. This is good, considering the fact that the improvement is achieved only by using an importance emphasis from the basic statistics (i.e., ef and nf) rather than other additional information.

RQ1: Our proposed approach enhanced all the SBFL formulas. The improvement of average ranks by our approach in the used benchmark was about **3** positions overall. In a few cases, the improvement even was more than **6** positions. In terms of average ranks, our approach reduced more positions. This indicates that using an importance weight could have a positive impact and enhances the SBFL results. Also, it encourages us to investigate other forms of importance weights in the future and measure their impacts on the effectiveness of SBFL.

It is worth mentioning that only using average ranks as an evaluation metric for SBFL effectiveness has its own set of drawbacks: (a) outlier average ranks could distort the overall information on the performance of any proposed approach. (b) it tells nothing about the distribution of the rank values and their changes before and after applying a proposed approach. Therefore, there is a more important category of evaluation than average ranks: improvements in the *Top-N ranks*, where the advantages are more obvious, as presented below.

6.2 Achieved improvements in the Top-N categories

According to [Kochhar et al. 2016] and [Xia et al. 2016], testers believe that examining the first five program elements in an SBFL ranking list is acceptable, with the first ten elements being the highest limit for inspection before the list is dismissed. Thus, the success of SBFL can also be measured by concentrating on these rank positions, which are collectively known as Top-N, as follows:

- **Top-1**: When a faulty program element is ranked first in the ranking list.
- **Top-3**: When a faulty program element's rank in the ranking list is less than or equal to three.
- **Top-5**: When a faulty program element's rank in the ranking list is less than or equal to five.
- **Top-10**: When a faulty program element's rank in the ranking list is less than or equal to ten.
- **Other**: When a faulty program element has a rank greater than ten in the ranking list.

Table 8 presents the number of bugs in the Top-N categories (cumulative) as well as their percentages for the entire dataset, before and after applying our proposed approach, as well as the differences between them. Here, improvement is defined as a decrease in the number of cases in the "Other" category and an increase in any of the Top-N categories.

It is evident that by relocating many bugs to higher-ranked categories, our proposed approach improved all Top-N categories. 4–11 bugs were moved from the "Other" category (with rank > 10) into one of the Top-N categories. This is significant since it raises the possibility of finding a bug with our approach while it was not very probable without it. This kind of interesting improvement is also known as *enabling improvements* [Beszédes et al. 2020]. Table 9 presents the summary of the enabling improvements achieved by our proposed approach.

It can be noted that each formula after applying our proposed approach achieves enabling improvements for at least 1% of the total number of single faults in the used dataset. In these cases, the basic SBFL formulas ranked the faulty method in the "Other" category, but our approach managed to bring it forward into the Top-10 (or better) categories. Note that, the formulas T*, O*, and BR* are the best in this aspect. Overall, each formula based on our

Table 8: Top-N categories

	Top-1		Top-3		Top-5		Top-10		Other	
	#	%	#	%	#	%	#	%	#	%
T	48	16.2	111	37.4	137	46.1	167	56.2	130	43.8
T*	59	19.9	125	42.1	148	49.8	178	59.9	119	40.1
Diff.	11	22.9	14	12.6	11	8.0	11	6.6	-11	-8.5
O	52	17.5	118	39.7	143	48.1	171	57.6	126	42.4
O*	57	19.2	122	41.1	147	49.5	176	59.3	121	40.7
Diff.	5	9.6	4	13.6	4	2.8	5	2.9	-5	-4.0
J	49	16.5	113	38.0	138	46.5	168	56.6	129	43.4
J*	55	18.5	121	40.7	143	48.1	172	57.9	125	42.1
Diff.	6	12.2	8	7.0	5	3.6	4	2.4	-4	-3.1
B	48	16.2	111	37.4	137	46.1	167	56.2	130	43.8
B*	52	17.5	118	39.7	143	48.1	171	57.6	126	42.4
Diff.	4	8.2	7	6.3	6	4.4	4	2.4	-4	-3.1
S	49	16.5	113	38.0	138	46.5	168	56.6	129	43.4
S*	55	18.5	121	40.7	143	48.1	172	57.9	125	42.1
Diff.	6	12.2	8	7.0	5	3.6	4	2.4	-4	-3.1
DS	51	17.2	103	34.7	127	42.8	151	50.8	146	49.2
DS*	53	17.8	105	35.4	129	43.4	155	52.2	142	47.8
Diff.	2	3.9	2	1.9	2	1.6	4	2.6	-4	-2.7
D	49	16.5	113	38.0	138	46.5	168	56.6	129	43.4
D*	55	18.5	121	40.7	143	48.1	172	57.9	125	42.1
Diff.	6	12.2	8	7.1	5	3.6	4	2.4	-4	-3.1
I	48	16.2	111	37.4	137	46.1	167	56.2	130	43.8
I*	52	17.5	118	39.7	143	48.1	171	57.6	126	42.4
Diff.	4	8.3	7	6.3	6	4.4	4	2.4	-4	-3.1
BR	48	16.2	111	37.4	132	44.4	166	55.9	131	44.1
BR*	59	19.9	123	41.4	147	49.5	174	58.6	123	41.4
Diff.	11	22.9	12	10.8	15	11.4	8	4.8	-8	-6.1
K	46	15.5	96	32.3	123	41.4	147	49.5	150	50.5
K*	51	17.2	103	34.7	127	42.8	151	50.8	146	49.2
Diff.	5	10.9	7	7.3	4	3.3	4	2.7	-4	-2.7
C	49	16.5	113	38.0	138	46.5	168	56.6	129	43.4
C*	55	18.5	121	40.7	143	48.1	172	57.9	125	42.1
Diff.	6	12.2	8	7.1	5	3.6	4	2.4	-4	-3.1

Table 9: Enabling improvements

	Rank > 10 (%) before our approach	Enab. improv. (%)
T vs. T*	130 (43.8%)	11 (3.7%)
O vs. O*	126 (42.4%)	5 (1.7%)
J vs. J*	129 (43.4%)	4 (1.3%)
B vs. B*	130 (43.8%)	4 (1.3%)
S vs. S*	129 (43.4%)	4 (1.3%)
DS vs. DS*	146 (49.2%)	4 (1.3%)
D vs. D*	129 (43.4%)	4 (1.3%)
I vs. I*	130 (43.8%)	4 (1.3%)
BR vs. BR*	131 (44.1%)	8 (2.7%)
K vs. K*	150 (50.5%)	4 (1.3%)
C vs. C*	129 (43.4%)	4 (1.3%)

proposed approach was able to achieve enabling improvements in the possible cases.

Higher categories have seen improvements as well, with 2–11 bugs moved to Top-1, for example. The percentages of bugs in each category before and after using the proposed method should be noted that they were computed based on the number of single faults in Defect4J, while the difference percentages were computed based on the number of single faults before applying our proposed approach.

There is also a non-accumulating form of Top-N categories that counts the cases where the bug fell in non-overlapping intervals of $[1]$, $(1, 3]$, $(3, 5]$, $(5, 10]$ or $(10, ...]$. These categories illustrate how often a bug is moved into a better (e.g., from $(5, 10]$ to $(1, 3]$) or worse (e.g., from $[1]$ to $(1, 3]$) category when using an SBFL technique. To put it another way, how many times do the bugs move to a higher-ranking category and how many times do they move to a lower-ranking category? As a result, an SBFL approach that improves all Top-N categories by moving a large number of bugs to higher-ranked categories performs better.

We may utilize the non-accumulating variation of these categories to better comprehend the actual changes across the different Top-N categories. This demonstrates whether the rank category has changed for the better. Table 10 presents these moves between the Top-N categories. The number of negative changes (worsening result) is indicated by the sign ✗. For example, suppose there were 10 bugs with a rank of more than 3 but less than or equal to 5 before we used our approach, but our approach produced a rank value larger than 5 (this is considered as a negative result). The number of positive changes is indicated by the sign ✓ (improving result). For example, if our approach resulted in a rank value less than 3 (this is considered as a positive result).

These numbers clearly show that improvement was dominant and degradation by our proposed approach was not observed in the used dataset, while we observed positive changes for 9–30 bugs.

Table 10: Top-N moves

	[1] ↓ ✗	(1,3] ↓ ✗	(1,3] ↓ ✓	(3,5] ↓ ✗	(3,5] ↓ ✓	(5,10] ↓ ✗	(5,10] ↓ ✓	Other ↓ ✓	✗	✓
T vs. T*	0	0	7	0	6	0	6	11	0	30
O vs. O*	0	0	4	0	4	0	3	5	0	15
J vs. J*	0	0	5	0	4	0	4	4	0	17
B vs. B*	0	0	4	0	4	0	4	4	0	16
S vs. S*	0	0	5	0	4	0	4	4	0	17
DS vs. DS*	0	0	2	0	2	0	1	4	0	9
D vs. D*	0	0	5	0	4	0	4	4	0	17
I vs. I*	0	0	4	0	4	0	4	4	0	16
BR vs. BR*	0	0	7	0	3	0	10	8	0	28
K vs. K*	0	0	4	0	5	0	3	4	0	16
C vs. C*	0	0	5	0	4	0	4	4	0	17

RQ2: Overall, it can be said that there were noticeable improvements in terms of the Top-N categories with positive results (improvements in **9–30** cases). Also, we were successful in increasing the number of cases in which the faulty method was ranked first by **4–23%**. Another interesting finding is that in some cases we were able to achieve more than **3%** enabling improvement by moving **4–11** bugs from the "Other" category into one of the higher-ranked categories. These cases are now more likely to be discovered and then fixed than before.

7 THREATS TO VALIDITY

In software engineering, each experimental study has some threats to its validity. In this work, the following actions to avoid or mitigate the threats of validity were considered:

- Selection of evaluation metrics: to be certain that our findings and conclusions are correct, we selected well-known evaluation metrics (i.e., average ranks and Top-N categories) that have been utilized in prior studies too.
- Correctness of implementation: a code review was performed to guarantee that our experiment implementation was correct. Furthermore, we have executed our approach multiple times to make sure it is implemented correctly.
- Selection of subject programs: we used Defects4J as a benchmark dataset in our study. Therefore, our findings cannot be generalized to other Java programs. However, we believe that the programs of Defects4J are representative and contain real faults of varied types and complexity. Defects4J is also extensively utilized in other fault localization research.
- Exclusion of faults: due to technical limits, we had to eliminate 5 faults from the Defects4J dataset (about 2% of the total number of single faults). The question is whether or not other researchers working with the same dataset will be able to reproduce our results. Our findings were not influenced in any way by this exclusion and the excluded faults were scattered almost uniformly throughout the dataset, thus we believe that this threat is very low.
- Selection of SBFL formulas: we used a collection of well-known SBFL formulas in our experiment to evaluate the effectiveness of our proposed approach, which represents only a small percentage of the reported formulas in the literature. The results demonstrate that all of them have improved. However, we cannot guarantee that using other different formulas would yield the same results. We used the formulas which are extensively used in other software fault localization research to limit the effect of this issue.

8 CONCLUSIONS

This paper proposes the use of emphasis on the failing tests that execute the program element under consideration in SBFL. We rely on the intuition that if a code element gets executed in more failed test cases compared to the other elements, it will be more suspicious, and it will be given a higher ranking. This is achieved by multiplying the initial suspicion score, computed by underlying SBFL formulas, of each program method by an importance weight that represents the rate of executing a method in failed test cases.

The main features of the proposed approach are: (a) it can be applied to a wide range of SBFL formulas without modifying a formula's structure or its concept. (b) it does not require any additional execution information rather than the basic statistics extracted from program spectra. (c) it solves the issue of an unbalanced SBFL matrix in the sense that there are much more passing tests than failing ones, and many SBFL formulas treat passing and failing tests similarly.

The experimental results of this study show that by shifting several bugs to the highest Top-N ranks, our approach improved the average ranks for all investigated formulas.

In the future, we would like to assess the effectiveness of our approach at different levels of granularity, such as the statement level. Involving other SBFL formulas in the study to identify which formulas give the best results and categorizing them according to that into groups would be interesting for further investigation. We also would like to employ other contextual/importance weights beyond method executions in failed test cases and to determine how they affect the SBFL's efficiency.

REFERENCES

R. Abreu, P. Zoeteweij, and A. J. C. Van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC)*. 39–46.

Rui Abreu, Peter Zoeteweij, and Arjan J. C. Van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*. 89–98.

R. Abreu, P. Zoeteweij, and A. J. C. Van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. 88–99.

Adekunle Akinjobi Ajibode, Ting Shu, and Zuohua Ding. 2020. Evolving Suspiciousness Metrics From Hybrid Data Set for Boosting a Spectrum Based Fault Localization. *IEEE Access* 8 (2020), 198451–198467.

Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A Learning-to-Rank Based Fault Localization Approach Using Likely Invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 177–188.

Babak Bagheri, Mohammad Rezaalipour, and Mojtaba Vahidi-Asl. 2019. An Approach to Generate Effective Fault Localization Methods for Programs. In *International Conference on Fundamentals of Software Engineering*. 244–259.

Arpad Beszédes, Ferenc Horváth, Massimiliano Di Penta, and Tibor Gyimóthy. 2020. Leveraging Contextual Information from Function Call Chains to Improve Fault Localization. In *IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 468–479.

Jiajun Jiang, Ran Wang, Yingfei Xiong, Xiangping Chen, and Lu Zhang. 2019. Combining Spectrum-Based Fault Localization and Statistical Debugging: An Empirical Study. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 502–514.

J. A. Jones, M. J. Harrold, and J. Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*. 467–477.

René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, 437–440.

Jeongho Kim, Jonghee Park, and Eunseok Lee. 2015. A new hybrid algorithm for software fault localization. In *Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication*. 1–8.

Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' Expectations on Automated Fault Localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 165–176.

Xia Li and Lingming Zhang. 2017. Transforming Programs and Tests in Tandem for Fault Localization. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 92 (Oct. 2017), 30 pages.

Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A Model for Spectra-Based Software Diagnosis. 20, 3 (2011).

Jonghee Park, Jeongho Kim, and Eunseok Lee. 2014. Experimental Evaluation of Hybrid Algorithm in Spectrum based Fault Localization. *International conference on Software Engineering Research and Practice (SERP)* (2014).

Qusay Idrees Sarhan and Árpád Beszédes. 2022. A Survey of Challenges in Spectrum-Based Software Fault Localization. *IEEE Access* 10 (2022), 10618–10639. https://doi.org/10.1109/ACCESS.2022.3144079

Qusay Idrees Sarhan, Béla Vancsics, and Árpád Beszédes. 2021. Method Calls Frequency-Based Tie-Breaking Strategy For Software Fault Localization. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 103–113. https://doi.org/10.1109/SCAM52516.2021.00021

Thorvald Julius Sørensen. 1948. A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on Danish commons. *København: I kommission hos E. Munksgaard*. (1948), 1–34.

Bela Vancsics, Ferenc Horvath, Attila Szatmari, and Arpad Beszedes. 2021. Call Frequency-Based Fault Localization. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 365–376.

W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2014. The DStar Method for Effective Software Fault Localization. *IEEE Transactions on Reliability* 63, 1 (2014), 290–308.

W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (aug 2016), 707–740.

Xin Xia, Lingfeng Bao, David Lo, and Shanping Li. 2016. "Automated Debugging Considered Harmful" Considered Harmful: A User Study Revisiting the Usefulness of Spectra-Based Fault Localization Techniques with Professionals Using Real Bugs from Large Systems. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 267–278.

Xiaofeng Xu, Vidroha Debroy, W. Eric Wong, and Donghui Guo. 2011. Ties within fault localization rankings: Exposing and addressing the problem. *International Journal of Software Engineering and Knowledge Engineering* 21, 6 (2011), 803–827.

Yi-Sian You, Chin-Yu Huang, Kuan-Li Peng, and Chao-Jung Hsu. 2013. Evaluation and Analysis of Spectrum-Based Fault Localization with Modified Similarity Coefficients for Software Debugging. In *2013 IEEE 37th Annual Computer Software and Applications Conference*. 180–189.

D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang. 2021. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Transactions on Software Engineering* 47, 2 (2021), 332–347.