Context Switch Sensitive Fault Localization

Ferenc Horváth
Department of Software Engineering
University of Szeged
Szeged, Hungary
hferenc@inf.u-szeged.hu

Roland Aszmann
Department of Software Engineering
University of Szeged
Szeged, Hungary
aszmann@inf.u-szeged.hu

Péter Attila Soha
Department of Software Engineering
University of Szeged
Szeged, Hungary
psoha@inf.u-szeged.hu

Árpád Beszédes
Department of Software Engineering
University of Szeged
Szeged, Hungary
beszedes@inf.u-szeged.hu

Tibor Gyimóthy
Department of Software Engineering
University of Szeged
Szeged, Hungary
gyimothy@inf.u-szeged.hu

ABSTRACT

Spectrum-Based Fault Localization (SBFL) is a popular technique to assist developers in pinpointing faulty elements within their code based on test outcomes and code coverage. In this paper, we examine the impact of context switching, i.e., when developers must frequently shift their attention between different code parts (such as methods and classes) while going down the SBFL ranked list to find the faulty statement. The basis of our study is the observation that it requires less effort to investigate statements that are next to each other rather than those in different methods and classes. In particular, we analyse the number of visited methods and classes, as well as the frequency of switches between them during the fault localization process. We found that, in programs from the Defects4J benchmark, developers need to explore 40 methods and 12 classes on average, before finding the faulty statement, leading to 53 method- and 40 class switches, respectively.

We introduce a novel context-aware metric that better approximates the total cost of finding a bug than traditional metrics that solely count the number of statements. Our metric considers both the statement number and the added cost of context switches. We also propose a new strategy for reducing the cost by optimizing the traversal of the elements in the ranked list based on the new context-aware metric. The algorithm not only lowers the number of statements that need to be investigated by 12% but also significantly reduces the number of class and method switches by 52%.

CCS CONCEPTS

• **Software and its engineering** → Object oriented languages; Software maintenance tools; Object oriented development; **Software testing and debugging**; **Software evolution**; **Maintaining software**.

KEYWORDS

Debugging, Spectrum-Based Fault Localization, SBFL cost metrics, Context switching.

ACM Reference Format:
Ferenc Horváth, Roland Aszmann, Péter Attila Soha, Árpád Beszédes, and Tibor Gyimóthy. 2024. Context Switch Sensitive Fault Localization. In *28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024), June 18–21, 2024, Salerno, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3661167.3661181

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EASE 2024, June 18–21, 2024, Salerno, Italy
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1701-7/24/06
https://doi.org/10.1145/3661167.3661181

1 INTRODUCTION

Finding the root cause of a bug in software is an inherently challenging and costly task, often requiring developers to navigate and comprehend complex and unknown codebases. This activity, often referred to as *Fault Localization (FL)*, is an inevitable part of the debugging process, which may contribute to as high as 50–75% of the overall development cost [5, 12, 36]. There are approaches to automate FL, and statistical analysis of program execution profiles and test case outcomes is a popular technique. The execution profile is known as the *Program Spectrum*, hence the name of the technique is Spectrum-Based Fault Localization (SBFL) [8, 18, 20, 28].

The program spectrum typically records the execution of the program at a particular *granularity level*, such as statements or methods. Since it is a statistical technique, no exact output is given about the faulty program location, but a ranked list of candidate elements. The technique is considered successful if the faulty elements are near the beginning of this list, so that it can provide useful aid to the programmer in the debugging process.

However, state-of-the-art algorithms are far from being optimal [19, 20]. The ranking list often includes non-faulty code elements before the faulty ones, hence the user has to spend valuable time examining irrelevant code (the amount of such examinations is called the *Expense* [3]). A significant issue with this is that the ranking list includes the code elements in a flat structure, either as a statement or a method list without any contextual or hierarchical information. The order of the elements can be arbitrary in a sense that no relationship between the subsequent elements is guaranteed. So the programmer may need to frequently switch between contexts when a new rank list element is examined. Or, we could also ask the question *if it is the only way to inspect the program elements following the ranked list in a linear fashion?*

Consider, for example, the output of an SBFL algorithm in Table 1 (execution of the DStar formula [27] on benchmark program Math, version 101 [16], statement level granularity). Here, the faulty element can be found at the 12th position in the ranked list. We can observe that the process starts with the method in which the faulty statement is located, *Parse*, but at steps three and five, we switch to different methods, before returning to the first one. The code line numbers after this point indicate that the examined statements are adjacent until reaching the faulty element. This part of the process is expectedly easier than the first couple of steps when the programmer had to switch between various methods.

These context switches can be very costly from program comprehension point of view [6, 19], which is very much required for successfully debugging [24, 36]. If the elements in the rank list would have more coherence to each other, it would be much easier for the programmers to work themselves down the list. An even more problematic aspect to this issue is that most state-of-the-art literature on SBFL uses trivial, overly optimistic measurement of Expense. Namely, by simply counting the number of elements in the rank list in front of the first faulty element, and use this number in various ways. This approach completely ignores the overhead of context switches, and we believe that this has serious consequences on assessing the actual effectiveness of particular approaches.

This paper explores the effects of context switching on fault localization to provide insights into how it affects the efficiency and effectiveness of SBFL. Furthermore, how this issue can be overcome by better measurement of the costs and improved rank list traversal that take into account context switching. More precisely, we have two goals with this research: 1) to define a more realistic Expense measurement that takes context switching into account, and 2) to improve how and in what order the ranked list of code elements is offered to the programmer that involves as few context switches as possible. Our contributions are the following:

(1) On the bug benchmark Defects4J, we measure how prevalent are context switches, and we find that 4–95 (40, on average) different methods are present in a single rank list in front of the faulty element.
(2) We define a new context aware measure for Expense assuming statement-level granularity. We find that the value of this metric is 2-10 times larger than the traditional Expense.
(3) The core of our approach is to better handle the order of methods corresponding to the statements in the rank list. Hence, we develop new method level ranking.
(4) Finally, we give algorithms for the optimized statement level rank listing which includes less context switching than the simple linear ordering of elements. We find 24-37% improvement in context-aware Expense with this approach.

2 BACKGROUND AND RELATED WORK

2.1 Spectrum-Based Fault Localization

Spectrum-Based Fault Localization is a family of statistical fault localization approaches, and in this work we concentrate on the most frequently used setup. The so-called "hit-based" spectrum [13] refers to the simple binary information if a code element (e.g., statement or function in a procedural, or method in an object-oriented context) is covered during the execution of a test case or not. The technique then determines the *suspiciousness score* of each program element on the chosen level of granularity based on the intuition that those code elements are more suspicious to contain a fault that are exercised by comparably more failing test cases than passing ones, while non-suspicious elements are traversed mostly by passing tests. Dedicated formulas are used to calculate the suspiciousness score, which in turn *rank the code elements* in score decreasing order, which is the output of the algorithm.

Table 1: Motivating example - DStar on Math version 101.

Rank	Class	Method	Line	Score
1	ComplexFormat	parse(ParsePosition)	349	0.33
2		parse(ParsePosition)	350	0.33
3		constructor()	58	0.29
4		constructor()	59	0.29
5		getDefaultNumberFormat()	237	0.29
6		parse(ParsePosition)	361	0.18
7		parse(ParsePosition)	364	0.18
8		parse(ParsePosition)	365	0.18
9		parse(ParsePosition)	374	0.18
10		parse(ParsePosition)	375	0.18
11		parse(ParsePosition)	376	0.18
12		parse(ParsePosition)	*377	0.18

A frequent implementation of SBFL is a binary coverage matrix (**C**) and a test results vector (**R**) as the basic data structures to calculate the suspiciousness scores for program elements [1]. In the coverage matrix, the rows represent the test cases and the columns are the program elements; statements or methods in our case. The value of a cell in the matrix is 1 or 0, depending on whether the code element is exercised by the test or not, respectively ($c_{t,e} \in \{0, 1\}$, where $e \in$ code elements and $t \in$ tests). An element in the results vector is 0 if test t passed, otherwise it is 1 ($r_t \in \{0, 1\}$).

All basic SBFL formulas rely on four fundamental statistics that are calculated from the spectrum. For each element e, the following sets are obtained, whose cardinalities are then used in the formulas:

e_{ep}: set of passed tests covering e
e_{ef}: set of failed tests covering e
e_{nf}: set of failed tests not covering e
e_{np}: set of passed tests not covering e

A large number of different formulas using these basic statistics have been proposed [14, 28, 31]. For this paper, we selected three well-known formulas to experiment with, which are shown in Equation (1). These are among the best performing formulas according to several publications, but at the same time are sufficiently different to foster diversity in the experiments.

$$\begin{aligned} &\textit{DStar}\ [27]\text{: } score(e) = \frac{|e_{ef}|^2}{|e_{ep}| + |e_{nf}|} \\ &\textit{Ochiai}\ [3]\text{: } score(e) = \frac{|e_{ef}|}{\sqrt{(|e_{ef}| + |e_{nf}|) \cdot (|e_{ef}| + |e_{ep}|)}} \\ &\textit{Barinel}\ [2]\text{: } score(e) = \frac{|e_{ef}|}{|e_{ef}| + |e_{ep}|} \end{aligned} \tag{1}$$

2.2 Measuring the Cost of Fault Localization

There are several complementary ways to measure the effectiveness of SBFL algorithms [14], but the basis for most of them is conceptually simple: we need to estimate (and minimize) the effort that is required from the programmers to find the faulty element with the help of the SBFL tool. The rank list provides a straightforward proxy for this property, so most previous approaches simply counted the number of elements in the list in front of the first faulty element, often collectively called the *Expense*. This approach makes the different spectrum formulas comparable to each other since the scores could not always be compared due to their different intervals. In this section, we overview the variants of the Expense metric.

Expense metric. The simplest version is the absolute Expense metric which means that we count the number of code elements in the rank list in front of the faulty one. One complication with this method are rank ties [32], i.e., situations when different code elements share the same suspiciousness scores. No formula can guarantee that all program elements can get different score values, either because of the construction of the formula, or because it is conceptually not possible (consider, for instance, statements in a basic block which always share the same scores). Typically, all elements in a rank tie are assigned the same rank value, based on one of the following approaches [29]: *minimum* (optimistic or the best case), where the top most; *maximum* (pessimistic or the worst case), where the bottom most; and the *average* strategy, where the medium position of the elements sharing the same suspiciousness value is used, respectively.

Equation (2) shows the *absolute average rank* calculation [3, 33], where i and f are code elements, the latter being the faulty one, while s_i and s_f are the respective suspiciousness score values.

$$E(f) = \frac{\left|\left\{i|s_i > s_f\right\}\right| + \left|\left\{i|s_i \geq s_f\right\}\right| + 1}{2} \tag{2}$$

Another issue is when the subject program contains more than one faulty element, which can happen very often. Here, typically the E value associated with the element with the highest suspiciousness score is used ($min(E(f))$, where $f \in$ {faulty elements}).

A normalized approach of the absolute average rank is the *EXAM score* [35] (also referred to as Wasted Effort [14]), which is the ratio of the absolute rank of the bug and the total number of ranked code elements, expressed as a percentage. This metric describes the percentage of elements that need to be reviewed to find the location of the bug. Often, elements with 0 score are excluded from the calculation. This metric enables comparing the effectiveness between different bugs and subject programs, but it has less relevance for understanding the actual required effort to find the bug. Since EXAM and absolute average rank result in the same order of efficiency, we discussed only the latter in our paper.

Accuracy. Several studies report that developers investigate only the first 5 or 10 elements in the recommendation (rank-)list by fault localization algorithms before giving up using the ranking [17, 30]. The family of metrics that distinguishes bugs where the minimum rank of faulty elements is less than or equal to N is commonly referred to as *Top-N* or *acc@N* [19]. This metric represents the number of successfully localized bugs within the top-n elements of the ranking lists. Higher values are better for this metric, and the typical values used for N are 1, 3, 5, and 10. Code elements ranked behind N, are referred to as the *Other* category.

2.3 Dealing with Context Switching

In related literature, we did not find any measurements of FL effectiveness that take into account the context switches. However, several works recognize the issue and try to address it differently.

Research done with programmers about their experience and expectations on SBFL tools usually highlight a critique that standalone code elements without contextual information are hard to examine. Such studies were published by Kochhar et al. [17] on user's expectations, by Parnin and Orso [19] and Souza et al. [23] about empirical studies with programmers, and by Horváth et al. [15] on results with users using think-aloud sessions.

Cheng et al. [6] provide a ranked list of program subgraphs made of program elements as the nodes (methods and blocks) and method call/control flow as edges. This represents contextual information for individual suspicious code elements. A similar approach has been proposed by Yan et al. [34], in which the initial ranking list is weighted based on the fault propagation context computed from the System Dependence Graph of the program. The GZoltar tool [10, 21] uses a hierarchy-based approach to group statements according to their code structure, and assigns for each code level (e.g., packages, classes, methods) the highest suspiciousness of its internal statements. This contextual information is implemented in an efficient graphical user interface that helps developers to navigate more efficiently between the code elements to locate the fault. The Jaguar tool [22] also employs a hierarchical view of the faulty code elements in the graphical user interface.

The work of De Souza et al. [9] is similar to our approach in the way that they also provide an order of methods in which the developers should examine the statements. The methods are selected by the highest suspiciousness score of their blocks and method call relationships, which provides the context for examining the code elements. Then, score-based filtering and examination step limiting is applied to reduce the number of code elements to be explored.

Despite one of their proposed methods, *CH*, is based on a similar idea to our algorithms, our approach significantly differs from their work. First, we provide an objective way to measure the context problem in the FL process, which enables the comparison of related approaches. We also give a set of different algorithms that use this context-aware effectiveness measure. They used only a small number of bugs from a handful of small programs with mixed seeded and real bugs, while we rely on a standard benchmark used in SBFL research with hundreds of manually verified real bugs. Finally, the basis of their approach are basic-blocks, while our algorithms work on statement-level data, which also makes direct comparison unfeasible.

3 STUDY SETTINGS

3.1 Benchmark

For the evaluation, we selected Defects4J (v1.4.0)[1], a widely used collection of Java programs and curated bugs in FL research [16]. This

[1]https://github.com/rjust/defects4j/tree/v1.4.0

benchmark contains six open-source Java projects with manually validated, non-trivial real bugs.

The original dataset contains 395 bugs, but there were cases which we had to exclude from the study due to various issues. We excluded those versions where no statements were marked as faulty, and where the faulty statements were not covered by any failing test. A total of 373 defects were included in the final dataset. Table 2 shows each project and its main properties. Columns 2-4 show program and test suite sizes, and the number of available bugs. The statistics related to the code elements and various levels of contexts are shown in Columns 5-7. The number of bugs that we used is presented in Column 8.[2]

Table 2: Main Properties of Programs Used from Defects4J (KLOC, Tests and No. Bugs Columns Data from [16])

Project	KLOC	Tests	No. bugs	Avg. no. statements	Avg. no. methods	Avg. no. classes	No. suitable bugs
Chart	96	2 205	26	4 003	653	36	25
Closure	90	7 927	133	16 222	3 540	418	125
Lang	22	2 245	65	809	138	6	62
Math	85	3 602	106	2 297	321	30	104
Mockito	20	1 379	38	1 727	665	114	30
Time	28	4 130	27	5 150	1 407	87	27
Total	**341**	**21 488**	**395**	**5 035**	**1 121**	**115**	**373**

3.2 Calculating Rank Lists

For the experiments in this paper, we used the code coverage and test results published by Pearson et. al [20]. To calculate the suspiciousness values of code elements, we rely on Ochiai, DStar and Barinel[3], as explained in Section 2.

Our implementation uses the average position tie breaking strategy (others include the best case and worst case, but we find that the average position is the most widely chosen strategy). We tackled the division by zero issue (which can affect many formulas) using the $c + \epsilon$ approach [26], where c is any coefficient in the formula and ϵ is the smallest representable floating number.

3.3 Measuring the Number of Context Switches

Equation (3) shows the four metrics that we use to express the number of context switches: the number of visited methods (V_M), the number of switches between methods (SW_M), the number of visited classes V_C, and the number of switches between classes (SW_C). These metrics serve as indicators to assess both the scale of methods and classes that developers must investigate during debugging, and the frequency at which context switches occur.

To calculate these metrics, we extracted the fully qualified name of the containing method for each statement. Next, we extended the basic file and line number based statement information in the ranked lists with class and method names. As a result, we could determine not only the statements that must be investigated, but the higher-level code elements too.

[2]The exact list of the bugs can be found in the online appendix.
[3]Ochiai, DStar and Barinel will be noted as *Och*, D^* and *Bar* in tables and figures in the next sections.

We define these metrics in similar fashion to the absolute average rank shown in Equation (2), thus i and f are statements (the latter being the faulty one), s_i and s_f are their score values, while $method(i)$ and $class(i)$ determine the containing method and class of i, and $method(0) = class(0) = nil$.

$$\begin{aligned} V_M(f) &= |\{method(i)|s_i \geq s_f\}| \\ V_C(f) &= |\{class(i)|s_i \geq s_f\}| \\ SW_M(f) &= |\{i|s_i \geq s_f \land method(i) \neq method(i-1)\}| \\ SW_C(f) &= |\{i|s_i \geq s_f \land class(i) \neq class(i-1)\}| \end{aligned} \tag{3}$$

3.4 Significance Testing

The ranking comparison results have been checked by testing statistical significance. Usually, a Wilcoxon sign-rank test [7] is used for this. However, in the context of SBFL, the test could encounter ties, i.e., when both approaches report the same rank for an element. To overcome this limitation, we used an implementation of the Wilcoxon test which copes with the ties by discarding the zero-differences. We complement the Wilcoxon test with the Cliff's Delta (d) effect size measure [11]. We used the typical thresholds of $d < 0.147$ "negligible", $d < 0.33$ "small", $d < 0.474$ "medium", otherwise "large" to determine the magnitude of effect sizes.

3.5 Research Questions

We formulate the following research questions in this paper:

RQ1 *How prevalent are context switches in traditional SBFL methods?* – We measure how big the search space is in terms of the number of investigated code elements on different granularity levels, and how frequently developers have to change between these elements.

RQ2 *To what extent does the cost of context switching affect the effectiveness of traditonal SBFL methods*? – We introduce a context-aware metric that accounts for the cost of context switching, and investigate how it relates to well-known Expense metrics.

RQ3 *How should method-level ranks be calculated from statement-level results?* – We investigate the trade-offs between different approaches to calculate method-level ranks.

RQ4 *Is there a more optimal approach to rank the statements that could minimize the number of context switches and how significant is the gain of such an approach in effectiveness?* – We introduce algorithms that use the raw ranking output of any SBFL technique and provide an alternative exploration order of code elements which is specifically optimized for minimizing the number of context switches.

4 PREVALENCE OF CONTEXT SWITCHES (RQ1)

The primary goal of our initial study is to quantify the extent to which context switching occurs during the FL process. We investigate this by examining four key metrics described in Section 3.3.

Table 3 depicts the results of our initial study regarding context metrics per program and aggregated over the whole benchmark. As a basis for comparison, columns 2-4 show the E values of the traditional statement-level SBFL algorithms, i.e., the average number

Table 3: Traditional Statement Expense and Context Metrics. Numbers in parentheses are the ratio of context metrics with respect to Expense.

	E			V_M			SW_M			V_C			SW_C		
	D*	Och	Bar	D*	Och	Bar	D*	Och	Bar	D*	Och	Bar	D*	Och	Bar
Chart	148.3	149.3	143.6	33.1 (0.22)	33.0 (0.22)	32.0 (0.22)	34.3 (0.23)	34.5 (0.23)	32.9 (0.23)	7.0 (0.05)	6.9 (0.05)	6.6 (0.05)	19.8 (0.13)	20.0 (0.13)	19.2 (0.13)
Closure	371.8	371.8	366.7	94.8 (0.26)	94.9 (0.26)	93.9 (0.26)	127.4 (0.34)	128.3 (0.35)	128.3 (0.35)	27.3 (0.07)	27.4 (0.07)	27.2 (0.07)	98.0 (0.26)	99.0 (0.27)	99.1 (0.27)
Lang	18.6	20.5	19.7	3.9 (0.21)	3.9 (0.19)	3.9 (0.20)	4.4 (0.23)	4.4 (0.21)	4.3 (0.22)	1.8 (0.10)	1.9 (0.09)	1.9 (0.09)	2.7 (0.14)	2.8 (0.13)	2.7 (0.13)
Math	63.0	63.0	63.3	10.0 (0.16)	10.0 (0.16)	9.8 (0.16)	11.0 (0.17)	11.0 (0.18)	10.9 (0.17)	4.0 (0.06)	4.0 (0.06)	3.9 (0.06)	7.6 (0.12)	7.6 (0.12)	7.6 (0.12)
Mockito	39.1	39.0	39.4	13.7 (0.35)	13.7 (0.35)	13.9 (0.35)	16.3 (0.42)	16.4 (0.42)	16.8 (0.43)	7.3 (0.19)	7.3 (0.19)	7.4 (0.19)	13.4 (0.34)	13.5 (0.35)	14.2 (0.36)
Time	98.3	99.0	99.1	26.4 (0.27)	26.5 (0.27)	26.4 (0.27)	34.2 (0.35)	35.9 (0.36)	36.7 (0.37)	9.0 (0.09)	9.0 (0.09)	9.1 (0.09)	26.6 (0.27)	28.3 (0.29)	29.1 (0.29)
Total	165.5	165.9	163.8	40.5 (0.24)	40.5 (0.24)	40.0 (0.24)	52.6 (0.32)	53.0 (0.32)	53.0 (0.32)	12.3 (0.07)	12.3 (0.07)	12.2 (0.07)	39.7 (0.24)	40.2 (0.24)	40.3 (0.25)

of statements to be examined before the faulty element. Context metrics are shown in the remaining columns, and the ratio of the actual context metric and the E value is shown in parentheses. For both methods and classes, the Lang program has the lowest values, while Closure has the highest. It is interesting to note that these extreme values are consistent with both the E values and the program sizes shown in Columns 5-7 in Table 2. However, this similarity is not present for the other programs. For example, Chart and Mockito are very similar in terms of average method count, but Chart's method-level context metrics are two to three times those of Mockito. Similarly, Math and Chart are close to each other in terms of average class number, but their class-level context metrics are also two to three times different.

It can also be observed that the number of visited methods is on average about a quarter of the E value, although this varies from 0.16 to 0.35 when examined program by program. In other words, for every 4 statements there is a new method in the rank list. The number of method switches is even higher, on average about 30%. Looking at larger contexts, i.e. classes, we find that the average ratio is only around 0.07, but in this case the number of switches is on average more than three times the number of classes visited. However, looking at this from the other perspective, every 14th statement in the list will belong to a new class.

Answer to RQ1: The average number of visited methods varies from 4 to 95, and the overall average is 40, which is about the quarter of the E value. In addition, the number of method switches is about 30% higher than this. At class level, the number of visited items varies between 2 and 27 (overall average is 12), which is approximately the tenth of the E value, while the number of switches is more than three times larger than this. This means that for every 4 statements there is a new method, and for every 14 statements a new class appearing in the rank list, what we believe is a high rate.

5 CONTEXT SWITCH AWARE EXPENSE(RQ2)

To answer RQ2, we introduce a new context-aware efficiency metric (E_{ctx}) based on the classic efficiency metrics for evaluating the performance of FL. This metric incorporates the cost of context switching on different granularity levels into the overall efficiency value. We argue that the required effort of investigating consecutive statements from the ranked list is much less than investigating those that are in separate methods or classes. Intuitively, upon examination of a single statement in a new context, the surrounding context must also be examined to some extent. Otherwise, it can be difficult to determine the correctness of the statement itself.

$$\begin{aligned} E_{ctx} = & |S| * \alpha_{s,U} + \\ & (V_M - 1) * \alpha_{m,U} + (SW_M - V_M) * \alpha_{m,V} + \\ & (V_C - 1) * \alpha_{c,U} + (SW_C - V_C) * \alpha_{c,V} \end{aligned} \tag{4}$$

Equation (4) shows how to calculate E_{ctx} based on the ranked list of statements (S), the corresponding methods and classes that were defined in Equation (3), and the α parameters. These parameters assign different factors to the elements based on whether the current context has been already investigated ($\alpha_{m,V}$ and $\alpha_{c,V}$) or not ($\alpha_{m,U}$ and $\alpha_{c,U}$) by the developer. The α parameters can be configured arbitrarily, which allows us to assign different weights to context switches on different granularity levels. Note that, when $\alpha_s = 1$ and all other parameters are zero, then $E_{ctx} = E$. Table 4 shows how we configured the α parameter of E_{ctx} for this experiment. We created three sets of values, Low, Medium and High to get a general idea about what effects the choice of α has on the overall performance. These values are somewhat optimistic, but we believe that values that follow the practice more closely would be even larger, however further research is needed to determine these parameters (see our discussion on the topic in Section 8).

Table 4: Configuration of the α Parameter of E_{ctx}

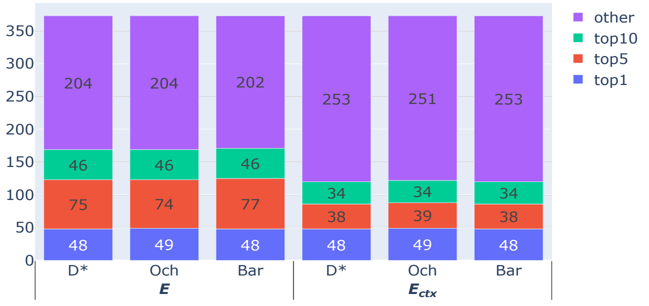
Context	V – Visited			U – Unseen		
	Low	Med.	High	Low	Med.	High
s – statement	–	–	–	1	1	1
m – method	1.5	2	3	2	4	9
c – class	2	4	9	4	8	27

In Table 5, we present the results of our measurements of E and E_{ctx} on Defects4J. As we can see, E_{ctx} values follow the increase of α parameters. Thus, naturally, E_{ctx} values are consistently larger than E values, due to the introduction of the α parameter. Again, the two extremes are Lang and Closure. However, the peak ratios of 2.70-2.76, 4.34-4.44 and 9.87-10.10 can be found at Mockito, while the minimal ratios align with the minimum values at Lang. Note that, while Mockito has lower E values than Math, the order is reversed when E_{ctx} is taken into account. Furthermore, the values of both metrics are fairly similar across the three formulas and configurations. On the per-program basis, E_{ctx} does not significantly modify the order of formulas, except for Mockito, where DStar and Ochiai switch places. However, across the whole benchmark, DStar slightly outperforms Barinel as α increases.

Figure 1 represents the accuracy metrics from the traditional and the context-based viewpoint. For the latter, we use the corresponding E_{ctx} values of the faulty statements, therefore, in this case Top-N could be considered as an effort-budget, where N represents the amount of developers' effort (including context switches) that could be spent on finding a bug. Note that, due to space limits, we present these results only with the Medium setting of α. Further results are available in the online-appendix. Top-1 results are the same, while, Top-5 and Top-10 values are significantly worse when we consider the cost of context-switches. However, it is natural for the number of context switches to increase as the number of elements to be analyzed increases. Also, it is important to mention that in this particular scenario, we utilized the traditional SBFL's rank lists and did not incorporate any optimizations for E_{ctx}.

Answer to RQ2: The cost of context switching has a notable effect on both the effectiveness and accuracy of SBFL approaches. We found that the context-aware metric values are 1.58 to 10.10 times larger than the traditional Expense values. Furthermore, accuracy significantly deteriorates in the Top-5 and Top-10 categories (by about 50% and 26%, respectively) when method and class contexts are taken into account.

Figure 1: SBFL's Accuracy in Terms of E and E_{ctx} Based Metrics



6 METHOD-LEVEL RANKING (RQ3)

In previous sections, we concluded that the amount of context switches is quite significant, so in the following we discuss how we can develop an algorithm for the traversal of suspicious code elements that minimizes the number of these switches, and hopefully the overall cost. Our initial focus is on optimizing the method level context switches, i.e., the ordering of the methods. Then, in Section 7 we concentrate on the statement traversal.

We consider two basic approaches for assigning suspiciousness scores and ranks to the methods. The first involves calculating method scores and ranks based on the method-level spectrum. This means that, in a matrix implementation, its columns contain methods and the coverage is computed at method-level (if a method is invoked, independent of what statements are exercised, it is considered covered). Then, the suspiciousness scores are computed using the spectrum formulas from this spectrum. Literature on SBFL dealing with method-level granularity typically follows this approach [4, 25, 37]. It has also the benefit of being more space and time efficient, which might be important in some applications.

The second approach involves calculating scores and ranks on statement-level, and then aggregating them to method-level using different aggregation functions such as the sum or maximum of statement-level scores in a method. While the first approach may be more straightforward and intuitive, the second approach provides a more comprehensive and accurate representation of method-level relations. Therefore, first we evaluate the trade-offs between these methods to determine which approach is more efficient.

For the first approach, we could have used method-level coverage matrix, but instead we started with the statement-level matrices, as they were already available, and aggregated the coverage to method-level (this leads to no practical difference). This aggregation is very simple: a method is considered covered by a test if at least one of its statements is covered. Next, we calculated the basic spectrum metrics, the scores, and the ranks for each method. The results are shown in Columns 2-4 of Table 6 (referred to as Matrix).

For the second approach, we started with the method scores that were calculated from the statement-level coverage using various aggregation functions based on the scores of a method's statements. We tested max, sum, mean, median, and other functions, but max and sum produced the best results. Here, the method is assigned a score which is the maximum or the sum of all the statements' scores in the method, respectively. We present the results of these two score-based approaches in Columns 5-7 and 8-10 of Table 6 (detailed results are available in the online appendix). The numbers in parentheses are the ratios compared to the Matrix version.

As can be seen, both score-based approaches outperform the traditional matrix-based one. Note that, Closure's baseline E values with the matrix-based approach are exceptionally large, which could make it an outlier, therefore we present the total values with and without Closure as well. Both approaches can achieve significant improvements, lowering the E values from 19.7-20.3 to around 11.8-12.0 and 7.0-8.2 on average, which is a 40-41% and 59-65% improvement compared to the matrix-based approach. On a per-program basis, the results are highly variable. The overall relative improvement is between 0.36 and 0.68 in case of the max and 0.15 to 0.54 in case of the sum function. The exception is the Chart program with the max function, where results are only slightly better (0.88-0.91) or worse (1.45) than the matrix-based approach. Considering the score-based approaches, sum yields better results than max with E values around 15.8-16.9 (7.0-8.2) and 23.5 (11.8-12.0), respectively. The only exception in this aspect is Mockito with the Ochiai formula. Note that, even though sum has a slight advantage here, we use max in Sections 7 and 8 as we found both methods to be approximately equally efficient in those applications.

On a per-program basis, compared to the matrix-based outcomes, not all results are statistically sound on the 0.05 significance level. The number of cases where the advantage of the score-based approaches is not clear according to the statistics is 11 out of the 36 total cases. This is the case with, for example, Chart, Mockito and Time with max and DStar, or Chart and Mockito with sum and Ochiai. However, in the other cases, and especially on the whole

Table 5: SBFL's Performance in Terms of E and E_{ctx} with different α parameters. Numbers in parentheses show the ratios to E.

	E			E_{ctx} (Low)			E_{ctx} (Medium)			E_{ctx} (High)		
	D*	Och	Bar	D*	Och	Bar	D*	Och	Bar	D*	Och	Bar
Chart	148.32	149.32	143.56	263.88 (1.78)	265.38 (1.78)	254.46 (1.77)	378.24 (2.55)	379.96 (2.54)	364.44 (2.54)	717.48 (4.84)	718.96 (4.81)	689.56 (4.80)
Closure	371.75	371.76	366.74	855.01 (2.30)	858.26 (2.31)	852.73 (2.33)	1305.70 (3.51)	1311.34 (3.53)	1304.35 (3.56)	2661.26 (7.16)	2672.86 (7.19)	2660.61 (7.25)
Lang	18.65	20.48	19.74	30.17 (1.62)	32.38 (1.58)	31.19 (1.58)	41.26 (2.21)	43.81 (2.14)	42.16 (2.14)	76.42 (4.10)	80.39 (3.92)	77.18 (3.91)
Math	63.03	62.98	63.35	101.73 (1.61)	101.77 (1.62)	101.52 (1.60)	139.43 (2.21)	139.54 (2.22)	138.63 (2.19)	260.28 (4.13)	260.52 (4.14)	257.02 (4.06)
Mockito	39.13	39.00	39.40	105.78 (2.70)	105.98 (2.72)	108.68 (2.76)	169.87 (4.34)	170.27 (4.37)	175.07 (4.44)	386.13 (9.87)	386.70 (9.92)	397.90 (10.10)
Time	98.33	99.00	99.11	228.09 (2.32)	234.61 (2.37)	237.70 (2.40)	350.11 (3.56)	360.85 (3.64)	365.93 (3.69)	725.56 (7.38)	746.44 (7.54)	756.56 (7.63)
Total	165.46	165.86	163.81	362.61 (2.19)	364.67 (2.20)	362.26 (2.21)	547.66 (3.31)	550.93 (3.32)	547.77 (3.34)	1108.78 (6.70)	1115.05 (6.72)	1109.10 (6.77)

benchmark, the score-based approaches perform statistically significantly better with a small to large effect size. Note, that results are omitted due to space limits, but detailed results can be found in the online appendix (see Section 9.1).

Answer to RQ3: We found that score-based aggregation approaches are much more effective than the matrix-based one. On average, they produce E values that are 0.16-0.91 times as much as the matrix-based approach's result. Moreover, the results are statistically significant, with a medium effect size on the entire benchmark and small-to-large effect sizes on a per-program basis in most cases.

7 CONTEXT-OPTIMIZED ALGORITHMS (RQ4)

Here, we introduce a set of algorithms that we designed to optimize the order in which program elements are examined, ultimately reducing the extra costs that are associated with context switches.

We leverage the results that were presented in the previous sections to help build the foundation for these approaches. Based on our observation in Section 4, it is evident that the occurrence of context switching at method-level is very notable. Section 5 further highlights the importance of addressing these changes to achieve cost efficiency. Therefore, reducing the frequency of these switches should be our top priority, and the findings from Section 6 about method-level outcomes will support our efforts to achieve this goal.

Our first algorithm takes a relatively straightforward approach. It determines the order in which to examine code elements through a two-step sorting process. First, it sorts the methods, i.e., the contexts, and then it sorts the statements within each method. The principles used for sorting at each level are alterable, but for the experiments with this algorithm, we sorted the methods in descending order based on their score values, which were aggregated by the max function (see Section 6). Statements within each method are also sorted in descending order based on their score values.

The algorithm suggests examining all statements in decreasing score order one method at a time. It starts with the method containing the statement with the highest score, and after all statements have been examined in this method it proceeds to the statements of the next method according to the method score list. Note that this algorithm skips instructions with no or zero score. In the following, we will refer to this algorithm as *ScoreSort*, or SS for short.

Our second algorithm is an extension of the first one. We noticed that in certain situations ScoreSort tends to stick to larger and seemingly more suspicious contexts. However, due to inherent inaccuracies in SBFL formulas that give us the basis of the aggregation and sorting steps, this has the potential to mislead the developers in identifying the faulty code elements. For instance, if a suspicious method does not contain the fault, according to ScoreSort, the programmer would still need to examine all statements in the method, even if there are statements with very low scores in it.

To mitigate this behavior, we introduce a bounding mechanism into the algorithm, which forces the algorithm to leave the method after examining a subset of its statements if they do not meet a certain score threshold. The algorithm retains flexibility by allowing a return to earlier contexts if the faulty element is not found. Although this may lead to a higher number of context switches, it has the potential to accelerate the process of finding the faulty statements by skipping the examination of statements with a small score value. With this bounding extension, we aim to find the balance between thorough context exploration and focused search.

Algorithm 1 shows the outline of our second algorithm, referred to as *ScoreSort-K* (SSK) in the following. In each iteration, a parameter called K is utilized to determine which statements are skipped during traversal. In practice, K is a list with progressively decreasing values, i.e., we allow successive iterations to examine statements with smaller and smaller scores. The algorithm begins by sorting methods in descending order of their scores (also using the max aggregation function). Then, for each K value, it iterates through methods, ordering statements within them by score and skipping those with scores lower than the current K value. If the fault is not found, the algorithm continues to the next iteration with a new K value. In our experiments, we set K to be the progressively decreasing list 40%, 30%, 20% 10% and 0% of the maximal score value in the faulty program version in question. Note that when $K = \{0\}$ then we get the basic *ScoreSort* algorithm.

Algorithm 1 ScoreSort-K

```
Require: $K$ the list of score thresholds
Require: $M$ the list of methods
  Sort($M$)
  $V \leftarrow \emptyset$                                   ▷ the set of examined statements
  for all $k \in K$ do
      for all $m \in M$ do
          Sort($m.statements$)
          for all $s \in m.statements$ do
              if $s \notin V \wedge s.score \geq k$ then
                  $V \leftarrow V \cup \{s\}$
                  if $s.buggy$ then return
```

Table 6: SBFL's Performance in Terms of E with the Matrix-based and the Score-based Approaches

	Matrix			Score (max)			Score (sum)		
	D*	Och	Bar	D*	Och	Bar	D*	Och	Bar
Chart	35.2	33.7	22.3	31.1 (0.88)	30.8 (0.91)	32.3 (1.45)	9.7 (0.28)	5.1 (0.15)	3.5 (0.16)
Closure	1311.4	1134.4	1127.8	48.6 (0.04)	48.5 (0.04)	48.0 (0.04)	36.0 (0.03)	34.7 (0.03)	34.7 (0.03)
Lang	5.3	5.9	6.5	3.6 (0.68)	3.5 (0.59)	3.8 (0.58)	2.9 (0.54)	2.7 (0.45)	2.5 (0.39)
Math	18.5	18.7	19.1	9.1 (0.49)	9.2 (0.49)	9.3 (0.49)	7.7 (0.42)	8.5 (0.46)	7.6 (0.40)
Mockito	26.5	26.8	35.5	13.3 (0.50)	13.2 (0.49)	12.8 (0.36)	12.6 (0.47)	15.4 (0.57)	11.0 (0.31)
Time	35.4	35.7	37.7	21.6 (0.61)	21.6 (0.61)	21.7 (0.58)	14.1 (0.40)	14.6 (0.41)	13.9 (0.37)
Total w/o Closure	19.7	19.8	20.3	11.8 (0.60)	11.8 (0.59)	12.0 (0.59)	8.0 (0.41)	8.2 (0.41)	7.0 (0.35)
Total	431.4	375.0	373.2	23.5 (0.05)	23.5 (0.06)	23.5 (0.06)	16.9 (0.04)	16.7 (0.04)	15.8 (0.04)

Table 7 shows how the optimized algorithms perform compared to the baseline i.e., the traditional SBFL approach with simple linear statement rank list (referred to as B in the following tables). Each block of the table corresponds to an algorithm, while columns are organized into three sets associated with the three types of metrics: E, E_{ctx} and the context metrics. The rows noted as "Diff" and "Diff %" represent the absolute and relative improvement of the actual algorithm relative to the traditional approach's results. Note that, due to space limits, we present these results only with the Medium setting of α. Further results are available in the online-appendix.

Context-metrics-based results show that both algorithms have a significant advantage over the baseline. The SS algorithm seems to have consistently better results in this aspect than SSK. It improves V_M by 36-37%, SW_M by 51-52%, V_C by 28% and SW_C by 52-53%. Considering E_{ctx}, on the whole benchmark, SS improves the baseline cost values by 36-37%, and SSK is slightly worse with an improvement of 24-33% on average. However, SSK proves to be better than SS in the case of the smallest program, i.e., Lang.

The E values show a much less clear picture. Although both algorithms can achieve 11-14% improvement when looking at the whole dataset, the Chart, Lang, Math and Mockito programs show that the new algorithms lag behind the traditional SBFL, even if only slightly. Overall, we can see the advantage of our algorithms even in the traditional E values, however, the really significant improvements are in E_{ctx} and the context metrics.

Table 8 depicts the Accuracy metrics of the optimized algorithms. Here, the focus is on the algorithms' performance in terms of E and E_{ctx}, not on the comparison of the metrics. Top-1 results are the same across all aspects. E-based results show that, on the contrary to previous results, SSK outperforms SS by a small margin, and SS is not able to improve on the baseline. In the Top-5 category, SSK almost matches the baseline, but it was able to localize slightly more bugs in the Top-10 category. However, the E_{ctx}-based results are completely different: SS is the best, particularly in the Top-5 category, SSK is second, and the baseline is last.

Based on the results of the statistical analyses described in Section 3.4, we found that statistically significant results were obtained at the 0.05 significance level for the whole dataset in all aspects. Examining the results separately, we obtain significant results in E values for Closure, Math and Mockito. In the case of E_{ctx}, Closure, Math and in some cases Time are the programs where the statistics confirm the results. For the context metrics, the situation is better, we got significant results in most cases. Note, that results are omitted due to space limits, but detailed results can be found in the online appendix (see Section 9.1).

Answer to RQ4: We proposed two approaches SS and SSK, that optimize the context switches and the overall cost. We found that, there were no significant differences between these algorithms in terms of efficiency. They can improve the context metrics by 21-52% on method level and 13-53% on class level. They also improve the overall cost of E_{ctx} by 24-37%, while matching or slightly improving the E values of the traditional SBFL approach. In addition, they have an advantage over SBFL in terms of accuracy metrics as well.

8 DISCUSSION

We argue that ignoring the cost of context switches along the ranking lists produced by SBFL algorithm could impair the fair evaluation of such approaches. To fill this gap, we introduce the context-aware Expense measurement approach. Since we used our own estimates for the costs of switches – the α parameters in Equation (4) –, this only provides a framework for a hopefully more accurate estimation of costs.

The parameters are determined based on our programming experience, and the rationale for using these values is that, in our view, the additional cost of the context switch when changing granularity levels increases exponentially. We agree with the suggestion by Souza et al. [9] that a fully accurate and general metric would require further experiments, mainly with humans, to establish and verify the final parameters of the metric. It is also needed to discover on which possible additional factors (e.g., code complexity) the parameters and the overall cost depend. The execution of a human study requires the careful consideration of various parameters and aspects such as the participants' experience, the diversity of the projects, etc. Despite these factors, the results obtained may still not be entirely generalizable. However, results with the different settings of α are expected to provide a reasonably accurate approximation of such experiments.

An interesting research direction and open question is how to handle the contexts, especially at different granularities, and what additional information can be included in the optimization. We have seen that reducing the number of context switches implies that the number of instructions to be examined increases slightly in some cases. Although the bounding mechanism in the ScoreSort-K

Table 7: Performance of the Traditional and Optimized Approaches in Terms of E, E_{ctx} and Context Metrics

		E			E_{ctx}			V_M			SW_M			V_C			SW_C		
		D*	Och	Bar	D*	Och	Bar	D*	Och	Bar	D*	Och	Bar	D*	Och	Bar	D*	Och	Bar
SSK	Chart	151.0	152.0	143.4	376.7	362.5	344.9	32.1	30.9	29.6	33.6	31.4	30.0	6.8	6.4	6.1	19.8	18.0	17.4
	Closure	301.4	316.1	295.0	802.3	946.5	802.6	56.6	71.3	57.0	63.7	79.6	64.0	18.9	23.1	19.2	49.2	62.0	50.1
	Lang	18.6	17.8	19.1	36.9	34.7	37.5	3.4	3.2	3.4	3.6	3.3	3.5	1.7	1.7	1.8	2.4	2.3	2.3
	Math	63.2	65.6	65.4	131.0	131.8	132.8	9.1	9.1	9.1	9.7	9.5	9.7	3.7	3.7	3.7	6.8	6.6	6.7
	Mockito	43.3	43.3	43.2	156.9	162.5	161.5	12.7	13.4	12.9	13.8	14.4	14.2	7.0	7.1	7.1	11.2	11.8	11.9
	Time	92.9	95.9	94.7	274.6	294.6	276.7	21.6	23.6	21.5	23.6	25.4	23.4	7.7	8.4	7.7	18.1	19.8	18.3
	Total	142.0	147.8	140.2	369.2	418.4	368.3	26.8	31.8	26.8	29.7	35.0	29.6	9.2	10.7	9.3	22.3	26.6	22.5
	Diff	-23.4	-18.1	-23.6	-178.4	-132.5	-179.4	-13.7	-8.6	-13.2	-22.9	-18.0	-23.4	-3.1	-1.6	-2.9	-17.4	-13.6	-17.8
	Diff %	-14.2	-10.9	-14.4	-32.6	-24.1	-32.8	-33.8	-21.3	-33.1	-43.5	-34.0	-44.1	-24.8	-13.3	-23.7	-43.9	-33.9	-44.1
SS	Chart	150.5	149.3	142.9	354.2	350.6	339.4	30.1	29.8	29.2	30.1	29.8	29.2	6.2	6.2	6.0	17.6	17.4	17.0
	Closure	304.4	304.1	302.4	749.1	747.9	739.8	54.6	54.4	53.4	54.6	54.4	53.4	18.1	18.0	17.9	41.5	41.5	41.0
	Lang	20.4	20.3	21.6	37.6	36.8	38.1	3.3	3.2	3.2	3.3	3.2	3.2	1.7	1.7	1.7	2.3	2.2	2.2
	Math	68.0	68.0	70.1	128.4	128.6	129.3	8.5	8.5	8.4	8.5	8.5	8.4	3.5	3.5	3.4	6.1	6.1	6.0
	Mockito	43.3	43.2	44.1	150.0	149.5	153.6	12.5	12.4	12.6	12.5	12.4	12.6	6.9	6.8	7.0	10.3	10.3	10.8
	Time	95.2	97.3	97.5	264.4	266.5	268.0	21.4	21.3	21.3	21.4	21.3	21.3	7.6	7.6	7.6	16.3	16.4	16.7
	Total	144.8	144.8	144.7	348.0	347.4	344.8	25.8	25.7	25.3	25.8	25.7	25.3	8.8	8.8	8.8	19.2	19.2	19.0
	Diff	-20.6	-21.1	-19.1	-199.6	-203.5	-203.0	-14.7	-14.8	-14.8	-26.8	-27.3	-27.7	-3.4	-3.5	-3.4	-20.6	-21.1	-21.3
	Diff %	-12.5	-12.7	-11.7	-36.5	-36.9	-37.1	-36.3	-36.5	-36.9	-51.0	-51.6	-52.3	-28.0	-28.3	-28.2	-51.7	-52.4	-52.8
B	Chart	148.3	149.3	143.6	378.2	380.0	364.4	33.1	33.0	32.0	34.3	34.5	32.9	7.0	6.9	6.6	19.8	20.0	19.2
	Closure	371.8	371.8	366.7	1305.7	1311.3	1304.4	94.8	94.9	93.9	127.4	128.3	128.3	27.3	27.4	27.2	98.0	99.0	99.1
	Lang	18.6	20.5	19.7	41.3	43.8	42.2	3.9	3.9	3.9	4.4	4.4	4.3	1.8	1.9	1.9	2.7	2.8	2.7
	Math	63.0	63.0	63.3	139.4	139.5	138.6	10.0	10.0	9.8	11.0	11.0	10.9	4.0	4.0	3.9	7.6	7.6	7.6
	Mockito	39.1	39.0	39.4	169.9	170.3	175.1	13.7	13.7	13.9	16.3	16.4	16.8	7.3	7.3	7.4	13.4	13.5	14.2
	Time	98.3	99.0	99.1	350.1	360.9	365.9	26.4	26.5	26.4	34.2	35.9	36.7	9.0	9.0	9.1	26.6	28.3	29.1
	Total	165.5	165.9	163.8	547.7	550.9	547.8	40.5	40.5	40.0	52.6	53.0	53.0	12.3	12.3	12.2	39.7	40.2	40.3

Table 8: Accuracy of the Traditional and Optimized Approaches in Terms of E and E_{ctx} Based Metrics

		E				E_{ctx}			
	Top -	1	5	10	Other	1	5	10	Other
SSK	D*	48	121	174	199	48	89	123	250
	Och	49	121	174	199	49	94	130	243
	Bar	48	125	175	198	48	91	125	248
SS	D*	48	121	162	211	48	101	133	240
	Och	49	121	163	210	49	103	136	237
	Bar	48	119	161	212	48	101	132	241
B	D*	48	123	169	204	48	86	120	253
	Och	49	123	169	204	49	88	122	251
	Bar	48	125	171	202	48	86	120	253

algorithm can compensate for this to some extent, the relationship between the changes in the two measures is not trivial. Also, a slight increase in the Expense could prove to be worthwhile when there is a significant decrease in context switches. The choice between SS, SSK, and other alternative algorithms could also depend on the environment in which they are used. For instance, if the current test suite or formula yields many items with small scores, it would be best to minimize the analysis of these. On the other hand, if we have many large contexts, including third-party code, we may decide that minimizing the number of switches is more important. Further complicating the situation, it is difficult to find a balance between the type of context to be given priority in the optimization and the attributes of the contexts to be used.

In this paper, we focus on methods, and sorting them by the maximum score of their statements. We also performed experiments based on the sum aggregation method, and we constructed class-level optimization algorithms. Moreover, we experimented with several settings of the K score thresholds, but these alternatives did not achieve similar efficiency as the two algorithms reported in the paper, so we publish these results only in the online appendix.

As already discussed, there are several other ways to deal with the context switches [6, 9, 34], measuring and optimising the cost is only one aspect of the issue. It may be useful, for example, to present contexts or context switches in a more informative and comprehensible way to the programmers. Additionally, it could also be possible to combine these methods in a complementary way.

It would also be interesting to experiment with contexts that are not tied to the existing structural levels of granularity. For example, statements that are in the same method, but relative far from each other could be considered as different contexts and the switching between them could be incorporated into E_{ctx} with a new parameter. It also remains future work, to investigate the influence of context-based metrics on methods that rely on SBFL, e.g., automatic program repair, fuzz testing, etc.

9 CONCLUSION

In this paper, we explored the phenomenon of context switching in the rank list of spectrum-based fault localization technique, and provided insights into how it affects the performance of SBFL algorithms and ultimately programmer's efficiency. To examine the frequency of context switches, we conducted a study on the Defects4J benchmark and found that, on average, the developers have to examine around 40 distinct methods before finding the faulty statement. We devised a new metric that accounts for context switches. Our analysis showed that this metric's value is 2-10 times greater than traditional Expense measurements. Finally, we showed that there are better ways to inspect code elements in the SBFL rank list and developed new strategies that focus on enhancing the order of methods associated with statements in the list. We showed that our algorithms significantly decrease context switching and improve context-aware Expense by 24-37% on average.

The perspectives of our research are to raise awareness to the deficiencies of current SBFL effectiveness assessment practices, and to encourage further research in the direction of making SBFL techniques more programmer oriented.

9.1 Data Availability

We provide the algorithm implementations, the measurement environment, the datasets and experiment results in an online appendix: https://zenodo.org/doi/10.5281/zenodo.11075509

ACKNOWLEDGMENTS

Project no TKP2021-NVA-09 has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme. Support by the the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. 2009. A Practical Evaluation of Spectrum-based Fault Localization. *J. Syst. Softw.* 82, 11 (nov 2009), 1780–1792. https://doi.org/10.1016/j.jss.2009.06.035

[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2009. Spectrum-based multiple fault localization. In *2009 IEEE/ACM Int. Conf. on Automated Softw. Engineering (CASE)*. IEEE, 88–99.

[3] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Proc. of the Testing: Academic and Industrial Conf. Practice and Research Techniques - MUTATION*. 89–98.

[4] 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 *Proc. of the 25th Intl. Sym. on Soft. Testing and Analysis (ISSTA)*. ACM, 177–188.

[5] Boris Beizer. 1990. *Softw. Test. Techniques.* Int. Thomson Computer Press, NY.

[6] Hong Cheng, David Lo, Yang Zhou, Xiaoyin Wang, and Xifeng Yan. 2009. Identifying Bug Signatures Using Discriminative Graph Mining. In *Proc. of the 18th Int. Sym. on Soft. Testing and Analysis (ISSTA '09)*. ACM, NY, USA, 141–152.

[7] William Jay Conover. 1998. *Practical nonparametric statistics.* Vol. 350. John Wiley & Sons.

[8] Higor Amario de Souza, Marcos Lordello Chaim, and Fabio Kon. 2016. Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges. *CoRR* abs/1607.04347 (2016). arXiv:1607.04347

[9] Higor A. de Souza, Danilo Mutti, Marcos L. Chaim, and Fabio Kon. 2018. Contextualizing spectrum-based fault localization. *Information and Software Technology* 94 (2018), 245–261.

[10] Carlos Gouveia, Jose Campos, and Rui Abreu. 2013. Using HTML5 visualizations in software fault localization. *Proceedings of 2013 1st IEEE Working Conf. on Softw. Visualization*, 1–10.

[11] Robert J Grissom and John J Kim. 2005. *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers.

[12] Brent Hailpern and Peter Santhanam. 2001. Software debugging, testing, and verification. *IBM Systems Journal* 41, 1 (Dec. 2001), 4–12.

[13] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. 1998. An empirical investigation of program spectra. In *Proc. of the 1998 ACM SIGPLAN-SIGSOFT workshop PASTE '98* (Montreal, Quebec, Canada). ACM, 83–90.

[14] Simon Heiden, Lars Grunske, Timo Kehrer, Fabian Keller, Andre Van Hoorn, Antonio Filieri, and David Lo. 2019. An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. *Software: Practice and Experience* 49, 8 (2019), 1197–1224.

[15] Ferenc Horváth, Árpád Beszédes, Béla Vancsics, Gergő Balogh, László Vidács, and Tibor Gyimóthy. 2022. Using Contextual Knowledge in Interactive Fault Localization. *Empirical Softw. Engineering* 27, 150 (2022), 69.

[16] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proc. of the 2014 Int. Sym. on Softw. Testing and Analysis.* ACM, 437–440.

[17] Pavneet S. Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proc. of the 25th Int. Sym. on Softw. Testing and Analysis - ISSTA 2016.* ACM, NY, USA, 165–176.

[18] Priya Parmar and Miral Patel. 2016. Software Fault Localization: A Survey. *Int. Journal of Computer Applications* 154, 9 (2016), 6–13.

[19] Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *Proc. of the 2011 Int. Sym. on Soft. Testing and Analysis* (Toronto, Ontario, Canada). ACM, 199–209.

[20] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. *Proc. of the 39th Int. Conf. on Softw. Eng.* (2017), 609–620.

[21] Alexandre Perez, André Riboira, and Rui Abreu. 2012. A Topology-Based Model for Estimating the Diagnostic Efficiency of Statistics-Based Approaches. In *2012 IEEE 23rd Int. Sym. on Softw. Reliability Engineering Workshops.* 171–176.

[22] Henrique Ribeiro, Roberto Andrioli de Araujo, Marcos Chaim, Higor Souza, and Fabio Kon. 2018. Jaguar: A Spectrum-Based Fault Localization Tool for Real-World Software. In *2018 IEEE 11th Int. Conf. on Softw. Testing, Verification and Validation (ICST).* 404–409.

[23] Higor Souza, Marcelo Lauretto, Fabio Kon, and Marcos Chaim. 2023. Understanding the use of spectrum-based fault localization. *J. of Softw.: Evolution and Process* (10 2023), e2622.

[24] Diomidis Spinellis. 2016. *Effective Debugging: 66 Specific Ways to Debug Software and Systems* (1st ed.). Addison-Wesley Professional. 256 pages.

[25] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Árpád Beszédes. 2022. Fault Localization Using Function Call Frequencies. *J. of Sys. and Softw.* 193 (2022), 111429.

[26] Dániel Vince, Attila Szatmári, Ákos Kiss, and Árpád Beszédes. 2022. Division by Zero: Threats and Effects in Spectrum-Based Fault Localization Formulas. In *Proc. of the 22nd IEEE Int. Conf. on Softw. Quality, Reliability, and Security (QRS'22)* (Guangzhou, China). 221–230.

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

[28] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Trans. Softw. Eng.* 42, 8 (2016), 707–740.

[29] W. Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. 2007. Effective Fault Localization using Code Coverage. In *31st Annual Int. Computer Softw. and Applications Conf. (COMPSAC)*, Vol. 1. 449–456.

[30] 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 Int. Conf on Softw. Maintenance and Evolution (ICSME).* IEEE, 267–278.

[31] Xiaoyuan Xie, Tsong Y. Chen, Fei-C. Kuo, and Baowen Xu. 2013. A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-based Fault Localization. *ACM Trans. Softw. Eng. Methodol.* 22, 4, Article 31 (Oct. 2013), 40 pages.

[32] Xiaofeng Xu, Vidroha Debroy, W. Eric Wong, and Donghui Guo. 2011. Ties within Fault Localization rankings: Exposing and Addressing the Problem. *Int. Journal of Softw. Eng. and Knowledge Eng.* 21 (2011), 803–827.

[33] Jifeng Xuan and Martin Monperrus. 2014. Learning to combine multiple ranking metrics for fault localization. In *2014 IEEE Int. Conf. on Softw. Maintenance and Evolution.* IEEE, 191–200.

[34] Yue Yan, Shujuan Jiang, Yanmei Zhang, Shenggang Zhang, and Cheng Zhang. 2023. A fault localization approach based on fault propagation context. *Information and Softw. Technology* 160 (2023), 107245.

[35] Abubakar Zakari, Sai Peck Lee, and Ibrahim A. T. Hashem. 2019. A single fault localization technique based on failed test input. *Array* 3 (2019), 100008.

[36] Andreas Zeller. 2009. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging* (2nd ed.). Morgan Kaufmann Publishers Inc., CA, USA. 424 pages.

[37] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. 2019. An empirical study of fault localization families and their combinations. *IEEE Trans. Softw. Eng.* 47, 2 (2019), 332–347.