Demystifying the Combination of Dynamic Slicing and Spectrum-based Fault Localization

Sofia Reis[1] , **Rui Abreu**[1] and **Marcelo d'Amorim**[2]
[1]IST, University of Lisbon & INESC-ID, Portugal
[2]Federal University of Pernambuco, Brazil
sofia.o.reis@tecnico.ulisboa.pt, rui@computer.org; damorim@cin.ufpe.br

Abstract

Several approaches have been proposed to reduce debugging costs through automated software fault diagnosis. Dynamic Slicing (DS) and Spectrum-based Fault Localization (SFL) are popular fault diagnosis techniques and normally seen as complementary. This paper reports on a comprehensive study to reassess the effects of combining DS with SFL. With this combination, components that are often involved in failing but seldom in passing test runs could be located and their suspiciousness reduced. Results show that the DS-SFL combination, coined as Tandem-FL, improves the diagnostic accuracy up to 73.7% (13.4% on average). Furthermore, results indicate that the risk of missing faulty statements, which is a DS's key limitation, is not high — DS misses faulty statements in 9% of the 260 cases. To sum up, we found that the DS-SFL combination was practical and effective and encourage new SFL techniques to be evaluated against that optimization.

1 Introduction

Software debugging is important and challenging. The task of locating the faulty code (i.e., fault localization) is particularly challenging. As such, countless automated techniques have been proposed in the past to reduce the cost of fault localization [Wong *et al.*, 2016]. Model-based software diagnosis (MBSD) [Reiter, 1987; de Kleer and Kurien, 2003] and Spectrum-Based Fault Localization (SFL) [Harrold *et al.*, 2000] are two popular techniques that leverage different principles to automate fault isolation [Abreu *et al.*, 2008]. At a high-level, MBSD techniques [Wotawa *et al.*, 2002; Mayer and Stumptner, 2008; Mayer *et al.*, 2008; Perez and Abreu, 2018; Ko and Myers, 2008] attempt to eliminate non-suspicious components whereas SFL techniques attempt to rank suspicious components with the goal of reducing fault diagnosis cost.

Dynamic Slicing (DS) [Agrawal and Horgan, 1990] is an instance of MBSD that has attracted huge attention in research over the last decades [Silva, 2012]. The technique traces back the statements in the code that influence a given point of interest, such as the evaluation of a failing assertion. Similarly to DS, SFL [Harrold *et al.*, 2000] received tremendous attention in research over the years [Wong *et al.*, 2016]. It computes suspiciousness values associated with program components (e.g., statements) based on coverage information gathered during the execution of test cases. More precisely, SFL uses coverage information of passing and failing test cases to identify likely faulty statements and produces on output a list of program components ranked in decreasing order of suspiciousness.

Intuitively, DS identifies irrelevant parts of the code (i.e., parts that do not contribute to the fault) whereas SFL ranks the relevant parts of the code. It is conceivable, therefore, to combine these two techniques based on the intuition that highly ranked statements (as per SFL), albeit covered by failing executions, could be in fact unrelated to the fault (as per DS). In fact, prior work reported promising preliminary results on this combination [Wotawa, 2010; Alves *et al.*, 2011; Hofer and Wotawa, 2012; Lei *et al.*, 2012; Guo *et al.*, 2018]. Unfortunately, they used a small set of subjects in their evaluation or over optimistic methods to evaluate fault localization improvement [Wu *et al.*, 2014; Lucia *et al.*, 2014b; Wen *et al.*, 2016].

Given the importance of fault localization, this paper revisits the problem of assessing the effectiveness of combining DS and SFL, addressing the key issues of prior work. We conducted a comprehensive study involving 260 faults from 5 different programs from the Defects4J benchmark [Just *et al.*, 2014], which is frequently used to evaluate fault localization research. Results show that DS misses faulty statements in 9% of the 260 faults analyzed. Furthermore, we found that the combination of the two approaches improves fault localization up to 73.7% (13.4% on average). To sum up, results indicate that the risk of applying the technique is relatively low for the positive impact it may bring; and, the tool implementing the technique works out-of-the-box, i.e., it puts no requirements on the running environment, subject programs it can be used, and requires no special setup.

The contributions of this work are:

1) An empirical study, using real-world applications and bugs, on the combination of DS-SFL for bug localization of *Java* faulty programs.
2) A tool[1], dubbed Tandem-FL, implementing the DS-SFL combination.

[1]Tool and dataset available at/through https://github.com/damorim/lithium-slicer (Accessed June 22, 2019)

2 Dynamic Slicing (DS)

Model-based diagnosis is a well-known approach that has been proposed by the DX community, a sub-field of AI that develops algorithms and techniques to determine the root-cause of observed failures. [Reiter, 1987; de Kleer and Kurien, 2003]. Applications of model-based diagnosis to localize software faults have demonstrated that it can be framed as dynamic slicing [Mayer and Stumptner, 2008; Mayer *et al.*, 2008].

Dynamic slicing [Agrawal and Horgan, 1990] is an instance of model-based software diagnosis that has been shown useful in automated software debugging where the region of interest is restricted to what can be reached from failing tests. Several dynamic slicing techniques exist. This paper uses Critical Slicing (CS) [DeMillo *et al.*, 1996] for its simplicity/generality. Critical Slicing prescribes a black-box language-semantics-agnostic recipe to computing executable slices. Critical Slicing simplifies the original program such that the resulting program preserves critical observations, such as assertion violations. More precisely, the simplification mechanism consists of deleting statements on the original program and checking if the output of the original and modified program are the same.

Our implementation of Critical Slicing is based on the Mozilla Lithium tool[2]. It takes as input a file and produces as output a simplified version of that file that satisfies a user-defined oracle. In our case, the oracle is defined such that the test produces the same failure manifestation as the one observed with the test execution on the original program. The Lithium minimization process starts by determining the initial size—in number of lines—of chunks to delete from the input file. For that, it chooses the highest power of two number smaller than the file size. For example, if the file has $1,000$ lines, Lithium sets the initial chunk size to 512 lines. Then, the tool starts a local search looking for chunks to exclude from the file. If the chunk satisfies the oracle it is removed. When no more chunks of that given size can be removed, Lithium divides the chunk size by two and repeats the search. This iterative process continues until no more lines can be removed. If n is the size of the input file and m is the size of the 1-minimal file found by Lithium, then Lithium usually performs $O(m \cdot \lg(n))$ iterations.

Claim 1 *The faulty statement may not be included in the critical slice of failing test cases.*

If test oracles are too general, it is possible, conceptually, that the critical slicing algorithm produces slices without the faulty code. Suppose that a test fails with a Null-Pointer Exception (NPE) and the criterion used to slice the code in that case (i.e., the oracle) is the presence of NPE in the output, regardless of the location that raises that exception. It is therefore possible that the slice obtained at some iteration of the aforementioned algorithm raises NPE, but it does so in a different part of the code. If that happens, the critical slicing algorithm would consider that as an acceptable simplification–as it satisfied the criterion–and would continue. As the algorithm does not backtrack, it would be impossible, to obtain a slice containing the faulty component from that point on. ■

It is worth noting that this is not a problem that afflicts only CS; *all* purely dynamic slicing techniques manifest this problem [Lin *et al.*, 2018]. To mitigate the issue in CS, it is necessary to make oracles more specific. For that, our approach was to use as the slicing criterion the stack trace associated with the test failure. That decision increases the chances that the sliced program will follow a path to the error similar to that followed by the original program. In addition, there are faults whose outputs may slightly differ from one execution to another. For example, exceptions including memory address identifiers, which change on each execution. For these cases, it was developed a mechanism to compare the rest of the output ignoring the identifier of the memory address.

3 Spectrum-based Fault Localization (SFL)

Spectrum-based fault localization is a statistical fault localization technique that takes as input a test suite including at least one failing test and reports on output a ranked list of components likely to be in fault [Wong *et al.*, 2016; Jones and Harrold, 2005; Lucia *et al.*, 2014a; Abreu *et al.*, 2009]. The following are given in SFL: a finite set $\mathcal{C} = \{c_1, c_2, ..., c_M\}$ of M system *components*[3]; a finite set $\mathcal{T} = \{t_1, t_2, ..., t_N\}$ of N system transactions, which correspond to records of a system execution, such as test cases; the error vector $e = \{e_1, e_2, ..., e_N\}$, where $e_i = 1$ if transaction t_i has failed and $e_i = 0$ otherwise; and an $N \times M$ coverage matrix $\mathcal{A}$, where $\mathcal{A}_{ij}$ denotes the coverage of component c_j in transaction t_i. The pair $(\mathcal{A}, e)$ is commonly referred to as spectrum [Harrold *et al.*, 2000]. Figure 1 shows an example spectrum.

$$\begin{array}{c|cccc|c}
\mathcal{T} & c_1 & c_2 & \cdots & c_M & e \\
\hline
t_1 & \mathcal{A}_{11} & \mathcal{A}_{12} & \cdots & \mathcal{A}_{1M} & e_1 \\
t_2 & \mathcal{A}_{21} & \mathcal{A}_{22} & \cdots & \mathcal{A}_{2M} & e_2 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
t_N & \mathcal{A}_{N1} & \mathcal{A}_{N2} & \cdots & \mathcal{A}_{NM} & e_N
\end{array}$$

Figure 1: An example spectrum.

Several types of spectra exist. The most commonly used is called hit-spectrum, where the coverage matrix is encoded in terms of binary *hit* (1) and *not hit* (0) flags, i.e., $\mathcal{A}_{ij} = 1$ if t_i covers c_j and $\mathcal{A}_{ij} = 0$ otherwise. SFL takes as input the pair $(\mathcal{A}, e)$ and produces on output a list of components ranked by their faulty suspiciousness. To that end, the first step of the technique consists of determining what columns of the matrix A resemble the error vector e the most. For that, an intermediate component frequency aggregator $n_{pq}(j)$ is computed $n_{pq}(j) = |\{i \mid \mathcal{A}_{ij} = p \wedge e_i = q\}|$. $n_{pq}(j)$ denotes the number of runs in which the component j has been active during execution ($p = 1$) or not ($p = 0$), and in which the runs failed ($q = 1$) or passed ($q = 0$). For instance, $n_{11}(j)$ counts the number of times component j has been involved ($p = 1$) in failing executions ($q = 1$), whereas $n_{10}(j)$ counts the number of times component j has been involved in passing executions. We then calculate similarity to the error vector by means of applying *fault predictors* to each component to produce a score quantifying how likely it is to be faulty. Components are then ranked according to such likelihood scores and reported

[2]Lithium details available at https://github.com/MozillaSecurity/lithium (Accessed June 22, 2019)

[3]A component can be any code artifact of arbitrary granularity such as a class, a method, or a statement [Harrold *et al.*, 2000].

to the user. Ochiai is one of those fault predictors that has shown to perform well [Wong *et al.*, 2016; Pearson *et al.*, 2017]. The Ochiai formula is given by the following equation *ochiai* $= n_{11}(j)/\sqrt{(n_{11}(j)+n_{01}(j))*(n_{11}(j)+n_{10}(j))}$.

4 The Tandem-FL Approach

This section describes Tandem-FL, a technique where DS and SFL work in tandem to solve the fault localization problem. To illustrate the idea behind Tandem-FL, consider a debugging scenario with five components $c_{1..5}$ and five transactions $t_{1..5}$, two of which are failing. Consider, additionally, that the component c_2 is faulty. Figure 2 shows, at the left-hand side of the arrow ($\Rightarrow$), an hypothetical spectra and its corresponding ranking, produced with the Ochiai predictor. Each line in the ranking shows, respectively, the rank/position, the component label, and the Ochiai score (in parentheses).

We illustrate the workflow of the Tandem-FL approach for this scenario in the following. Let us analyze components at the granularity of statements. First, Tandem-FL picks the first two statements at the top of the ranking, c_4 and c_3, for further analysis. (The cutoff point is user-defined.) Second, the technique picks a failing test to slice the code, say t_2. A slice is a set of statements. Let us assume that the slice obtained for the transaction t_2 is $\{c_2, c_5\}$, i.e., it excludes statements c_4 and c_3 that were previously covered by the test. Finally, the spectra and ranking are updated. The right-side of Figure 2 shows the modified spectra and ranking after slicing the code against t_2. It is worth noting that the technique slices the code for every failing tests. Intuitively, slicing enables the identification of statements in the spectra whose values can be set to zero, i.e., the components marked with 0 are irrelevant to determine the test output.

Let us now observe the effect of this spectra modification on the ranking. Analyzing the Ochiai formula, one observes that, for the components j, which are not part of the slice of a failing test, the combination reduces the value of $n_{11}(j)$ and increases the value of $n_{01}(j)$. Therefore, suspiciousness of those components decrease. Similar argument applies to other fault predictors. In the running example from Figure 2, the components c_3 and c_4, which are not in the slice of t_2, have their suspiciousness reduced, enabling the faulty component c_2 to rise from the third to the first position in the ranking.

Briefly, the workflow of Tandem-FL consists of four steps:

1. Compute spectra S and ranking R for the input test suite;
2. Select top k most suspicious classes, according to R;
3. Compute slicer for every suspicious file, obtained in Step 2, and every failing test;
4. Assemble all the resulting slicers of each failing test;
5. Adjust spectra S, from step 1, with the slices of each failing test, from step 4, and then recompute the ranking.

Claim 2 *The rank of faulty statements cannot decrease if the slice includes the faulty statements.*

The proof is based on the outlined argument that irrelevant components j have their ranks reduced as $n_{11}(j)$ decreases and $n_{01}(j)$ increases. If those irrelevant components appear at positions above the faulty component, it is possible that

$\mathcal{T}$	c_1	c_2	c_3	c_4	c_5	e
t_1	1	0	1	1	0	✓
t_2	0	1	1	1	1	✗
t_3	1	0	1	0	0	✗
t_4	0	1	0	0	1	✓
t_5	1	0	0	1	1	✓

$\Rightarrow$

$\mathcal{T}$	c_1	c_2	c_3	c_4	c_5	e
t_1	1	0	1	1	0	✓
t_2	0	1	0	0	1	✗
t_3	1	0	1	0	0	✗
t_4	0	1	0	0	1	✓
t_5	1	0	0	1	1	✓

(a) Spectra update.

1	c_4 (0.59)
2	c_3 (0.55)
3	c_2 (0.35)
4.5	c_1 (0.32)
4.5	c_5 (0.32)

$\Rightarrow$

1	c_2 (0.35)
2.5	c_1 (0.32)
2.5	c_5 (0.32)
4	c_3 (0.29)
5	c_4 (0.00)

(b) Ranking update.

Figure 2: Modifications on spectra and ranking as result of slicing code against test t_2. Double arrow ($\Rightarrow$) indicates before and after.

the faulty component becomes more suspicious relative to j, i.e., the ranking of the faulty component increases. If those components appear at positions below the faulty components, the ranking of the faulty components is unaffected. ■

5 Evaluation

We studied the following research questions:

RQ1. *How effective is DS in eliminating code?*

RQ2. *How often does DS miss faulty statements?*

RQ3. *How effective is Tandem-FL for bug localization?*

The first question evaluates the ability of dynamic slicing to reduce the size of application code. Improvements on fault localization depend on that ability. The second question addresses the problem that affects all dynamic slicing techniques [Lin *et al.*, 2018]—that of missing faulty statements (see Claim 1). Developers would not be able to successfully debug code for the cases that issue is manifested. The third research question evaluates the impact of the combination, as substantiated by Tandem-FL, to fault localization.

5.1 Objects of Analysis

We used subject programs from the Defects4J benchmark [Just *et al.*, 2014] in our evaluation. Table 1 shows size, number of tests, and number of faults, for each considered program. Two faults from Apache commons-math were not considered because we were unable to reproduce them in our environment. Apache commons-lang is a library providing utility functions for the `java.lang` API. Apache commons-math is a library of self-contained mathematics and statistics components. JFreechart is a Java library for creating charts. Joda-Time is a library for manipulating date and time. Mockito is a mocking testing framework.

As to quantify complexity of fault localization on this benchmark, we measured the frequency of "faults of omission" (FOO), which is a fault whose fix is materialized only with the addition of new code [Pearson *et al.*, 2017]. Intuitively, the ability of techniques to produce accurate diagnostic reports in those cases might be poor as existing statements are not problematic. We found that this dataset has a total of 59.7% of FOOs, making it challenging for techniques to perform well.

Project	Size (LOC)	# Tests	# Faults
Apache commons-lang	111,751	6,057	65
Apache commons-math	306,276	26,797	104
JFreechart	230,159	8,458	26
Joda-Time	141,610	3,289	27
Mockito	22,787	8,835	38

Table 1: Characterization of Defects4J subjects.

Project	$k = 5$	$k = 10$
Apache commons-lang	1.40	31.46
Apache commons-math	10.30	12.34
JFreechart	59.30	53.64
Joda-Time	17.27	32.02
Mockito	16.54	21.67

Table 2: DS reduction in file size (percentages). Higher is better.

5.2 Techniques

One important independent variable for this study is the scope of analysis for the techniques. Analyzing long rankings would be unacceptably expensive not only for humans [Parnin and Orso, 2011] but also for machines. For example, Critical Slicing needs to re-compile the affected file at every iteration, i.e., after deleting statements and before checking the oracle (see Section 2). We control the scope of analysis through the variable k, denoting the number of most-suspicious files that will be analyzed. These files are selected from the SFL ranking as follows. First, we build a ranking of files by determining, for each ranked statement, the file that declares it. Then, we select the top-k distinct files from such file ranking.

The techniques we evaluate in this study are Tandem-FL^k and SFL^k, which is the comparison baseline. Tandem-FL^k is as defined in Section 4 (see Step 2 of the workflow) whereas SFL^k is as SFL, but it produces a ranking only including the statements from the top-k ranked files. In this study, we used 5 and 10 as the values of k, following the same choice as in previous fault localization studies [Ang *et al.*, 2017].

5.3 Results and Discussion

This section discusses and answers the research questions.

RQ1: How effective is DS in eliminating code?

A necessary but insufficient condition for improvement of fault localization with Tandem-FL is that dynamic slicing eliminates a high amount of code from the best ranked files. Table 2 shows, as percentages, the size reduction of the top files obtained with dynamic slicing for different values of k. Overall, results indicate that dynamic slicing substantially reduces the size of highly-ranked files.

RQ2: How often does DS miss faulty statements?

A fundamental issue of purely dynamic slicing techniques is the risk of discarding faulty statements (see Section 2). This research question evaluates the practical impact of this conceptual issue. Table 3 shows the number of cases dynamic slicing captures at least one of the faulty statements among the sliced files for different values of k. These numbers indicate that in 87.3% of the cases, on average, dynamic slicing finds at least one faulty statement in the top 5 of the highest-ranked faulty classes. This number is slightly improved to 91.2% if we consider the top 10 of the highest-ranked faulty classes.

Project	$k = 5$	$k = 10$
Apache commons-lang	96.9	96.9
Apache commons-math	89.4	95.2
JFreechart	76.9	84.6
Joda-Time	81.5	85.2
Mockito	71.1	78.9
Total	87.3	91.2

Table 3: Tandem-FL^k performance on capturing faulty statements, as percentages. Higher is better.

	$k = 5$		$k = 10$	
Project	SFL	Tandem-FL	SFL	Tandem-FL
Apache commons-lang	84.6	96.9	84.6	96.9
Apache commons-math	81.7	89.4	85.6	95.2
JFreechart	84.6	76.9	92.3	84.6
Joda-Time	77.8	81.5	81.5	85.2
Mockito	63.2	71.1	71.1	78.9
Total	79.6	86.5	83.5	90.4

Table 4: Number of faults where at least one of the faulty statements appears at the report of the technique. Higher is better.

To analyze the impact of dynamic slicing in context, we also compared Tandem-FL^k and SFL^k considering the number of cases where the fault is captured by the technique within a certain bound k. Note that the reason SFL^k misses the fault is different compared to Tandem-FL^k. SFL^k captures the fault if it is included within the top k files; it misses the fault otherwise. Table 4 shows results, indicating that Tandem-FL^k, typically, captures more faults compared to SFL^k. In only two cases, highlighted in gray color, SFL^k outperformed Tandem-FL^k. These results show that the majority of faulty statements are found on the 5 highest ranked faulty classes and that Tandem-FL^{10} is the technique that performs better, missing only 23 out of the 260 (∼8.8%) faults.

RQ3: How effective is Tandem-FL for bug localization?

In **RQ3**, we evaluate the impact of combining DS with SFL to improve fault localization. The first experiment we ran consists of measuring the *difference of diagnosis cost* between the two fault localization techniques. This metric has been previously used in other studies [Wong *et al.*, 2016; Ang *et al.*, 2017; Pearson *et al.*, 2017; Perez and Abreu, 2018]. More specifically, we computed $\Delta C = C(\text{SFL}^k) - C(\text{Tandem-FL}^k)$, where C denotes the diagnosis cost and is obtained by computing the mean position in the ranking of all buggy statements. $\Delta C < 0$ means that Tandem-FL^k performs worse compared to its baseline. That could happen if dynamic slicing misses the faulty statement. $\Delta C = 0$ means that the faulty statement remained in the same position. $\Delta C > 0$ means that Tandem-FL^k outperformed the baseline, i.e. the mean position of the faulty statements is smaller in Tandem-FL^k compared to that of the baseline. It is worth noting that we assume perfect bug

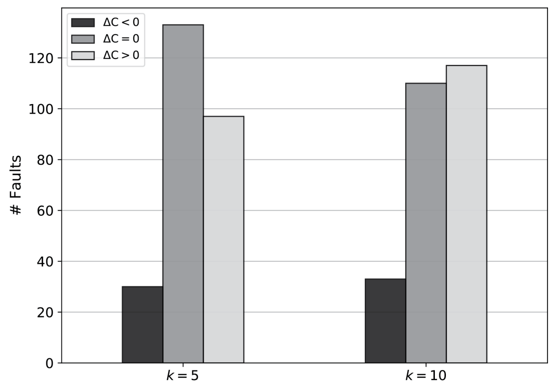

Figure 3: Delta Cost of Diagnosis (ΔC) per k

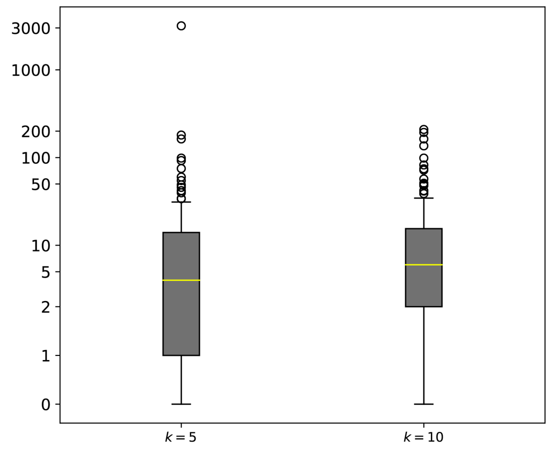

Figure 4: Distributions of ΔC considering all cases where Tandem-FL^k outperformed the baseline

understanding [Parnin and Orso, 2011], i.e., if the developer sees the bug in the ranking, he is able to precisely determine if it is the faulty one. Figure 3 summarizes the performance of Tandem-FL^k relative to SFL^k. The figure shows a histogram over ΔC for different values of k. Note that (1) the number of cases where the baseline outperforms Tandem-FL^k is small (e.g., 33 of the 260 faults for $k = 10$) and that (2) the number of cases Tandem-FL^k outperforms the baseline is significative (e.g., 117 of the 260 faults for $k = 10$). One of the following reasons justifies the baseline outperforming Tandem-FL^k: (1) a fault of omission or (2) the test has design limitations which DS is not capable of handling (mentioned in Section 2).

Figure 4 shows the distributions of ΔC considering all the cases where the predicate $\Delta C >= 0$ holds, i.e., it focuses only on the cases where Tandem-FL^k outperformed the baseline technique in at least one k. To sum, although we observed a relatively small number of cases where ΔC is negative (33) and a high number of cases where ΔC is zero (110), there is a considerable amount of cases where ΔC is positive and for those cases Tandem-FL^k seems very beneficial.

In the following, we present statistical analysis of the techniques. Table 5 reports descriptive statistics about the distributions of C for the techniques under different values of k. In bold, we present the results for the 227 faults corresponding to the cases where $\Delta(C) \geq 0$. The number in parentheses are relative to the dataset including all faults. We observe the following when considering the faults which can benefit from Tandem-FL (numbers in bold). For $k = 5$, Tandem-FL achieves a mean improvement of 41.1 (9.57% on average) statements on the ranking and an impressive maximum ranking improvement of 3171 (96.7% on average) statements. For $k = 10$, Tandem-FL^k achieves a mean improvement of 19 (13.35% on average) statements on the ranking and a maximum ranking improvement of 209 (73.7% on average) statements. Such difference is justified by the fact that entire files are discarded in Tandem-FL^5, whose statements are included in the ranking of Tandem-FL^{10}. Consequently, the improvements obtained by the technique will not be as high if slicing on these additional files is unable to aggressively and safely discard statements. This also shows that slicing files, in addition to slicing statements from those files, is very beneficial for fault localization performance.

	$k = 5$		$k = 10$	
	SFL	Tandem-FL	SFL	Tandem-FL
Mean	**779.3** (584.3)	**738.2** (679.7)	**630.6** (516.4)	**611.9** (614.0)
Median	**187.7** (49.4)	**167.2** (69.3)	**167.2** (53.0)	**107.5** (56.9)
Variance	**1218.3** (1223.1)	**1186.3** (1352.4)	**1058.4** (1148.1)	**1048.6** (1294.9)
Shapiro-Wilk	W = 0.67 p = 9.64 x 10^{-15}	W = 0.66 p = 5.59 x 10^{-15}	W = 0.63 p = 1.09 x 10^{-15}	W = 0.63 p = 8.95 x 10^{-16}
Friedman	χ^2 = 175.18 p-value = 9.71 x 10^{-38}			

Table 5: Statistical tests for C

Table 5 presents the statistics to determine if the observed results are statistically significant. Shapiro-Wilk tests the null hypothesis that results are drawn from a normal distribution. The test is performed for all techniques and values of k. With 99% confidence, the results tell us that the distributions are not normal. Given that C is not normally distributed, we used Friedman, a non-parametric statistical test of hypothesis. The null hypothesis is that the rankings obtained with all techniques and variants are the same. With 99% confidence, the results show that the distributions are distinct. To understand which techniques perform differently, i.e., to answer the question *Does Tandem-FL^k perform differently than SFL^k*, we performed a Siegel post-hoc analysis. Figure 5 reports results. Each square shows the statistical significance of the difference in diagnostic accuracy amongst the different techniques–the lighter color means no statistical significance and smaller values of p indicates higher significance. Based on these results, it is possible to determine with 95% confidence that the performance of each pair of techniques is different (except for Tandem-FL^5 versus Tandem-FL^{10}, where there is no significance).

As the results are statistically significant, it is possible to state that Tandem-FL^k has a positive effect on the localization of some types of bugs. There is indeed a problem on using slicing on fault of omission bugs that needs further research. Although, this dataset has more than 50% of fault of omissions, our simple slicing approach was capable of improving the ranking of 95 faults for $k = 5$ and 115 faults for $k = 10$. These may be promising results from what Automated Pro-

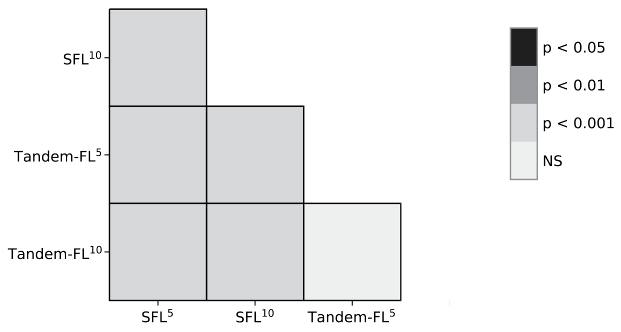

Figure 5: Siegel post-hoc analysis results

gram Repair (APR) techniques may directly benefit to improve the accuracy of their own techniques.

5.4 Threats to Validity

A potential threat to *external validity* relates to the set of programs used in our study. When choosing the projects for our study, our aim was to opt for projects that resemble a general and large-sized application. To reduce selection bias and facilitate the comparison of our results, we decided to choose a benchmark that is popular in the community [Just *et al.*, 2014]. Our study did not evaluate the faults from the Closure compiler project because of the high CPU cost of running Tandem-FL^k on it. A potential threat to *construct validity* relates to the choice of oracle we used for the Lithium slicer. We found empirically that our choice was able to capture the faults for the cases we analyzed. The main threat to *internal validity* lies in the complexity of several of the tools used in our experiments, most notably the Lithium toolset, the SFL diagnosis tool, and the implementation of Tandem-FL. To mitigate this threat, we carefully inspected our code and looked for discrepancies and incoherence in our results.

6 Related Work

Model-based diagnosis is a well-known approach that has been proposed by the DX community, a sub-field of AI [Reiter, 1987; Wotawa, 2002; de Kleer and Kurien, 2003]. Applications of model-based diagnosis to localize software faults have demonstrated that it can be framed as dynamic slicing [Mayer and Stumptner, 2008; Mayer *et al.*, 2008; Nica *et al.*, 2013]. Not many interesting techniques have been pursued due to scalability limitations. We have tackled this using the program slicing technique — critical slicing [DeMillo *et al.*, 1996] — for its simplicity and generality.

Dynamic Slicing has found its main application in fault localization [Agrawal and Horgan, 1990], e.g., on the minimization of failing tests through the identification of code components that do not contribute to the fault revelation. Research in this area has mainly focused on slicing code efficiently [Wang and Roychoudhury, 2008; Wang and Roychoudhury, 2004] and locate and contour faults of omission [Zhang *et al.*, 2007; Lin *et al.*, 2018].

Statistics-based techniques (e.g., [Pearson *et al.*, 2017]) are popular automated fault localization techniques. Spectrum-based fault localization (SFL) is amongst the most common statistical fault localization technique used to rank faulty components [Wong *et al.*, 2016; Jones and Harrold, 2005; Lucia *et al.*, 2014a; Abreu *et al.*, 2009]. This technique uses tests coverage information to rank program's components.

Prior work has investigated the DS-SFL combination [Wotawa, 2010; Alves *et al.*, 2011; Hofer and Wotawa, 2012; Lei *et al.*, 2012; Guo *et al.*, 2018; Christi *et al.*, 2018]. Overall, these studies reported promising results but based their conclusions on a very small group of (mostly artificial) faults and using different techniques. In this study, a larger set of faults is evaluated and a more rigorous experimental methodology. In contrast with other studies on this combination, our approach is the first using Critical Slicing to minimize test cases and leveraging SFL rank to choose and minimize the top-k of highest-ranked faulty classes.

7 Conclusions and Future Work

Several approaches have been proposed in the literature to reduce software debugging costs through automated fault diagnosis with the goal of improving productivity in software development. In this domain, Dynamic Slicing (DS) and Spectrum-based Fault Localization (SFL) are very popular techniques and normally seen as complementary.

This paper reports the outcome of a comprehensive study using real-world applications with the goal of demystifying the impact of combining DS with SFL. The results of our empirical study show that, in practice, DS misses faulty statements infrequently 9% (23 misses in 260 cases) and that the DS-SFL combination, coined as Tandem-FL^k, improves the diagnostic accuracy up to 73.7% (13.4% on average).

Despite tacit opinions of the research community about the usefulness of DS in automated debugging, we found the DS-SFL combination practical (as per the relatively low number of misses) and effective (as per the improvements of $\Delta(C)$ on the relevant cases), and encourage new SFL techniques to be evaluated against that optimization. Future work includes 1) improving our oracle heuristics to handle more exceptions, 2) conducting experiments in more Defects4J faults and other datasets, and 3) handling cases of faults of omission.

Acknowledgments

This material is based upon work supported by Fundação para a Ciência e a Tecnologia (FCT), with the reference PTDC/CCI-COM/29300/2017, and by national funds also through FCT with reference UID/CEC/50021/2019. This research was partially funded by INES 2.0, FACEPE grants PRONEX APQ 0388-1.03/14 and APQ-0399-1.03/17, and CNPq grant 465614/2014-0.

References

[Abreu *et al.*, 2008] Rui Abreu, Alberto González, Peter Zoeteweij, and Arjan J. C. van Gemund. Automatic software fault localization using generic program invariants. In *SAC*, 2008.

[Abreu *et al.*, 2009] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *JSS*, 2009.

[Agrawal and Horgan, 1990] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *PLDI*, 1990.

[Alves *et al.*, 2011] Elton Alves, Milos Gligoric, Vilas Jagannath, and Marcelo d'Amorim. Fault-localization using dynamic slicing and change impact analysis. In *ASE*, 2011.

[Ang *et al.*, 2017] Aaron Ang, Alexandre Perez, Arie van Deursen, and Rui Abreu. Revisiting the practical use of automated software fault localization techniques. In *ISSREW*, 2017.

[Christi *et al.*, 2018] Arpit Christi, Matthew Lyle Olson, Mohammad Amin Alipour, and Alex Groce. Reduce before you localize: Delta-debugging and spectrum-based fault localization. In *ISSREW*, 2018.

[de Kleer and Kurien, 2003] Johan de Kleer and James Kurien. Fundamentals of model-based diagnosis. *IFAC Proc. Volumes*, 2003. SAFEPROCESS.

[DeMillo *et al.*, 1996] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. Critical slicing for software fault localization. In *ISSTA*, 1996.

[Guo *et al.*, 2018] Anbang Guo, Xiaoguang Mao, Deheng Yang, and Shangwen Wang. An empirical study on the effect of dynamic slicing on automated program repair efficiency. In *ICSME*, 2018.

[Harrold *et al.*, 2000] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software: Testing, Verification and Reliability*, 2000.

[Hofer and Wotawa, 2012] Birgit Hofer and Franz Wotawa. Spectrum enhanced dynamic slicing for better fault localization. In *ECAI*, 2012.

[Jones and Harrold, 2005] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, 2005.

[Just *et al.*, 2014] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *ISSTA*, 2014.

[Ko and Myers, 2008] Andrew J. Ko and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *ICSE*, 2008.

[Lei *et al.*, 2012] Yan Lei, Xiaoguang Mao, Ziying Dai, and Chengsong Wang. Effective statistical fault localization using program slices. In *COMPSAC*, 2012.

[Lin *et al.*, 2018] Yun Lin, Jun Sun, Lyly Tran, Guangdong Bai, Haijun Wang, and Jinsong Dong. Break the dead end of dynamic slicing: Localizing data and control omission bug. In *ASE*, 2018.

[Lucia *et al.*, 2014a] Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process*, 2014.

[Lucia *et al.*, 2014b] Lucia, David Lo, and Xin Xia. Fusion fault localizers. In *ASE*, 2014.

[Mayer and Stumptner, 2008] Wolfgang Mayer and Markus Stumptner. Evaluating models for model-based debugging. In *ASE*, 2008.

[Mayer *et al.*, 2008] Wolfgang Mayer, Rui Abreu, Markus Stumptner, Arjan JC Van Gemund, et al. Prioritising model-based debugging diagnostic reports. In *DX*, 2008.

[Nica *et al.*, 2013] Mihai Nica, Simona Nica, and Franz Wotawa. On the use of mutations and testing for debugging. *Software: practice and experience*, 2013.

[Parnin and Orso, 2011] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, 2011.

[Pearson *et al.*, 2017] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *ICSE*, 2017.

[Perez and Abreu, 2018] Alexandre Perez and Rui Abreu. Leveraging qualitative reasoning to improve sfl. In *IJCAI*, 2018.

[Reiter, 1987] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 1987.

[Silva, 2012] Josep Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys*, 2012.

[Wang and Roychoudhury, 2004] Tao Wang and Abhik Roychoudhury. Using compressed bytecode traces for slicing java programs. In *ICSE*, 2004.

[Wang and Roychoudhury, 2008] Tao Wang and Abhik Roychoudhury. Dynamic slicing on java bytecode traces. *TOPLAS*, 2008.

[Wen *et al.*, 2016] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. Locus: Locating bugs from software changes. In *ASE*, 2016.

[Wong *et al.*, 2016] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *TSE*, 2016.

[Wotawa *et al.*, 2002] Franz Wotawa, Markus Stumptner, and Wolfgang Mayer. Model-based debugging or how to diagnose programs automatically. In *IEA/AIE*, 2002.

[Wotawa, 2002] Franz Wotawa. On the relationship between model-based debugging and program slicing. *Artificial Intelligence*, 2002.

[Wotawa, 2010] Franz Wotawa. Fault localization based on dynamic slicing and hitting-set computation. In *QSIC*, 2010.

[Wu *et al.*, 2014] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. Crashlocator: Locating crashing faults based on crash stacks. In *ISSTA*, 2014.

[Zhang *et al.*, 2007] Xiangyu Zhang, Sriraman Tallam, Neelam Gupta, and Rajiv Gupta. Towards locating execution omission errors. In *PLDI*, 2007.