On the Influence of Multiple Faults on Coverage-Based Fault Localization

Nicholas DiGiuseppe
University of California, Irvine
Department of Informatics
ndigiuse@ics.uci.edu

James A. Jones
University of California, Irvine
Department of Informatics
jajones@ics.uci.edu

ABSTRACT

This paper presents an empirical study on the effects of the quantity of faults on statistical, coverage-based fault localization techniques. The former belief was that the effectiveness of fault-localization techniques was inversely proportional to the quantity of faults. In an attempt to verify these beliefs, we conducted a study on three programs varying in size on more than 13,000 multiple-fault versions. We found that the influence of multiple faults (1) was not as great as expected, (2) created a negligible effect on the effectiveness of the fault localization, and (3) was often even complimentary to the fault-localization effectiveness. In general, even in the presence of many faults, at least one fault was found by the fault-localization technique with high effectiveness. We also found that some faults were localizable regardless of the presence of other faults, whereas other faults' ability to be found by these techniques varied greatly in the presence of other faults. Because almost all real-world software contains multiple faults, these results impact the use of statistical fault-localization techniques and provide a greater understanding of their potential in practice.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Experimentation, Reliability

Keywords

Fault Localization, Debugging, Empirical Studies

1. INTRODUCTION

An unfortunate, yet virtually inevitable activity of software development is a long and costly debugging process. One component of debugging is localizing the fault, or *fault localization.* Studies have shown fault localization to be the most difficult and time-consuming component of debugging [18]. To reduce this cost, researchers have developed automated fault-localization techniques. One category of automated fault-localization techniques can be described as dynamic, execution-based, statistical fault localization (e.g., [1, 9, 11, 13]). In this paper we will refer to such techniques as *coverage-based fault localization* (CFL). A common characteristic of these techniques is that they attempt to identify features of the faulty software whose execution correlates with software failure.

Despite years of incremental improvements to CFL techniques, our current understanding of some fundamental properties of the effectiveness of these techniques remains primitive. Since its inception, many research endeavors have explored how to improve current CFL techniques and find new ways to automate fault-localization. Yet CFL has suffered from a lack of thorough examination and much guess work regarding its utilization. Currently much research has been conducted to compare differing CFL techniques' effectiveness, but not much attention has still be brought to exploring how and why CFL techniques are affected by the quantity of faults in a program.

The purpose of this work is to identify how CFL is affected by the quantity of faults within a program. Previous work has made claims that CFL is no longer effective when more than one fault is present (see Section 2), yet real-world programs will almost always include more than one fault. As such, for CFL techniques to be useful for real-world programs and faults, they need to be effective in the presence of multiple faults or be augmented with additional techniques, such as failure clustering.

In this paper, we investigate the ability for CFL techniques to effectively localize faults in programs containing many faults. We explicitly investigate the scenario where a developer is iteratively finding and fixing faults that are causing failures. As such, our study focuses on the localizability of *any* fault, as opposed to the localizability of a *particular* fault that is causing one particular failure. The results of our study provide evidence that in general, regardless of fault quantity, CFL techniques are able to effectively localize at least one fault, and as such, debugging can be performed with the assistance of such automated tools, iteratively, until failure-free. Our study shows that for up to ten faults, CFL continues to be effective and in many cases even improves in effectiveness with fault quantity.

While investigating the underlying factors that informed our results in this study, we discovered a phenomenon that we call fault-localization interference. This phenomenon hinders a fault's ability to be localized in the presence of other faults and is found to be prominent in software containing multiple faults. We conducted an evaluation to study fault-localization interference and its impact on CFL. The study showed that this type of interference was prevalent, and demonstrated that despite this prevalence, the CFL technique persisted in being effective for at least one fault.

The main contributions of the paper are

1. The results of an empirical analysis of CFL that challenges and in many cases refutes the commonly held belief regarding CFL effectiveness, while at the same time finding evidence of the factors that caused researchers to believe otherwise. Our results show that CFL techniques can be effective, regardless of quantity of faults, thus potentially leading to a savings of time and costs for debugging when used in practice. These results can help developers by providing them with accurate information about how multiple faults affect their ability to perform automated fault localization.

2. A description and analysis of how faults can interfere with each others' localizability. We refer to this interference as *fault-localization interference*, and present studies and results to help characterize it and gauge its prevalence. We show that fault-localization interference is prevalent, yet rarely has a significant effect on the ability to localize at least one fault. The results of these studies may aid understanding of fault interaction, time estimation, and regression testing.

3. We provide an analysis of our results that have implications for the applicability of automated techniques such as CFL techniques and failure-clustering techniques as a precursor for CFL. These results and their implications can inform decisions of which automated techniques are needed for an organization, while considering their trade-offs.

In the next section we present a thorough background which motivates these studies. In Section 3 we explain our experiment's design and present its results. In Section 4 we analyze those results and their implications. In Section 5 we identify threats to validity of these studies. Finally, in Section 6, we conclude and discuss future work.

2. BACKGROUND AND MOTIVATION

To provide the necessary background to motivate this work, we overview CFL techniques (Section 2.1), summarize current perceptions of CFL techniques in the presence of multiple faults (Section 2.2), provide an example that demonstrates the assumptions for these perceptions (Section 2.3), define the interaction of multiple faults with regard to CFL (Section 2.4), and describe the need for further study on this topic (Section 2.5).

2.1 Coverage-based Fault Localization

Many approaches have been proposed which perform coverage-based fault-localization. Jones and colleagues [8, 9] proposed a fault-localization technique, TARANTULA, which utilizes whole test suites (or any subset thereof) to infer likely locations for faults based upon the relative participation of the passing and failing test cases and the events that occurred during execution. This work originally was presented to target instruction-level coverage. Other researchers provided additional ideas for performing such inferencing. For example, Liblit and colleagues proposed Statistical Bug Isolation, which monitors and utilizes randomly sampled subsets of coverage in order to reduce the runtime overhead of full instrumentation for fault localization [11]. Their work targeted predicates, including branching, function return values, and dynamic invariants. Liu and Han proposed a technique, called SOBER, that can utilize branch *profiles* (as opposed to coverage) with the goal of achieving greater accuracy and precision for localization [13]. Abreu and colleagues investigated the use of alternate inferencing metrics, and found that an existing metric borrowed from the biological sciences community outperformed some of the earlier techniques [1]. Yilmaz and colleagues proposed utilizing time profiles of methods to infer fault locations in the code [19]. Artzi and colleagues localize faults in dynamic web applications by combining variations of the TARANTULA technique with a mapping between instructions in the program and the fragments of output they produce [2].

The main insight of each of these CFL techniques is that, when running a test suite, execution events that correlate with failures are more likely to be the cause (i.e. fault or bug) of those failures. Said differently, events that occur mostly in failing test cases, but rarely in passing test cases, are more *suspicious* of being the fault. This inferencing examines the event similarities among the failing test cases and differentiates those similarities from the events occurring in the passing test cases.

With this explanation — event similarities among failing test cases — we begin to understand the belief that these types of CFL techniques will not perform adequately for programs with multiple faults. In a program that contains multiple faults, the similarities among failures — failures that are each caused by different faults — might not correspond to either fault. Instead, the similarities may correspond to other, non-fault-relevant code, and thus, may mislead the inferencing technique and user of the technique.

2.2 Perceptions of CFL for Multiple Faults

The second author of this paper, with his colleagues, have written about this intuition in prior work. For example, Jones and colleagues [9] reported that the "effectiveness of the technique declines on all faults as the number of faults increases" (they also note that these results may be misleading and require further study). In [10], Jones and colleagues investigated the use of failure clustering to remove "noise" caused by one fault inhibiting the localization of another.

Furthermore, this intuition was not exclusively ours: several researchers have made such claims. For instance, Denmat and colleagues state that the TARANTULA technique (and thus other similar CFL techniques) makes implicit hypotheses requiring independence of multiple faults — every failure is caused exclusively by a single fault — and when these hypotheses do not hold, the technique does not provide "good results" [6]. Zheng and colleagues developed specialized techniques targeting programs containing multiple faults because, in the presence of multiple faults, traditional CFL techniques "cannot distinguish between useful bug predictors and predicates that are secondary manifestations of bugs" [20]. Referring to coverage-based fault-localization techniques, Srivastav and colleagues stated that "multiple faults in a software many times prevent debuggers from effi-

	t1	t2	t3	t4	*suspiciousness*	*suspiciousness (excluding t3)*	*suspiciousness (excluding t4)*
`if ( b ) {`	•	•	•	•	0.7	0.6	0.6
`   bug 1;`	•		•		0.7	0.0	0.7
`} else {`		•		•	0.7	0.7	0.0
`   bug 2;`		•		•	0.7	0.7	0.0
`}`	•	•	•	•	0.7	0.6	0.6
pass/fail?	P	P	F	F			

Figure 1: Example code snippet containing two faults. This example demonstrates the possibility of multiple faults creating noise that interferes with fault localization effectiveness.

ciently localizing a fault" [17]. Debroy and Wong state that "incorrect matching of failed test to fault, ... may in turn result in poor fault localization" [5].

Such assertions are not unreasonable or unfounded. Indeed, studies (e.g., [9, 10, 20]) have shown that for *specific* faults, the presence of other faults may mask the ability of CFL techniques from properly localizing them. It is this motivation — localizing multiple specific faults — that motivates the field of failure clustering (e.g., [10, 12, 15, 20]).

2.3 Example

Consider, for example, the code presented in Figure 2.2. The program snippet listed in the first column contains two faults, labeled "`bug1;`" and "`bug2;`." The next four columns list the test cases: t1 and t2 are passing test cases, and t3 and t4 are failing test cases. Test cases t3 and t4 fail due to different faults, `bug1` and `bug2`, respectively. In the next three columns are the suspiciousness scores, varying only the subset of the test suite used. The suspiciousness score is calculated with the Ochiai metric, which was proposed by Abreu and colleagues to augment the TARANTULA technique. In recent work, Abreu and colleagues [1] proposed the Ochiai coefficient, which originated in the molecular biology domain to augment the TARANTULA technique. The equation for Ochiai can be represented as

$$suspiciousness(i) = \frac{failed(i)}{\sqrt{totalfailed * (failed(i) + passed(i))}} \quad (1)$$

where *passed*(i) is the number of passed test cases in which instruction i is executed, *failed*(i) is the number of failed test cases in which i is executed, and *totalfailed* is the number of failed test cases in the test suite.

The first column shows the suspiciousness scores when considering all test cases in the test suite. When utilizing all test cases, the *Suspiciousness* of all five instructions is 0.7. Because all instructions are equally suspicious, the technique is ineffective at localizing any faults — in other words, it did not reduce the search space to find the fault. However, if we exclude test case t3 from the test suite and suspiciousness calculation, the technique successfully localizes `bug2`. Likewise, if we exclude test case t4, the technique successfully localizes `bug1`. In this example, the technique was rendered ineffective at localizing either fault, when both faults caused test failures, due to the fact that different test failures were caused by different faults. This example validates the intuition that multiple faults *can* cause interference with each other thus inhibiting the effectiveness of the inferencing technique.

2.4 Fault-localization Interference

The example in Figure 2.2 demonstrates the ability of the presence of a fault to interfere with the localization of another fault. We define the term *fault-localization interference* as the phenomenon of the presence of a fault to cause the ineffectiveness of the fault-localization technique to locate another fault. While some quantity of ineffectiveness might be disregarded by developers as insignificant, however we denote *any decrease* in fault-localization effectiveness due to the additional presence of another fault as fault-localization interference. For the example in Figure 2.2, if either `bug1` or `bug2` were removed, the other would have been localized more effectively (as evidenced by the results when excluding the test case failures that they caused, respectively). Thus, for this example, each fault provided fault-localization interference for the other.

2.5 Motivation for Further Study

The fault-localization interference presented in the example is a particular type of interference — one in which the interference prevents any faults from being localized effectively. While limited, the results of the case study presented in [9] showed that often the interference caused some faults to be obscured — that is, made less localizable with the CFL technique — while others (usually the faults causing the interference) were highly localizable. One goal of this study is to determine the prevalence and nature of fault-localization interference. In other words, *how often does fault-localization interference occur, and when it does, how often does it take the form that causes ineffective localization for all faults?*

In the presence of multiple faults, even with fault-localization interference, if the CFL technique is effective at localizing at least one fault, the technique can be useful. Past studies have examined the localizability of all faults and used poor localization of any individual fault as evidence of the need for additional techniques such as failure clustering. However, in general, developers can neither be assured of the presence of multiple faults nor the quantity of them. Hence, debugging tends to be an iterative process — failures are used to find and fix faults, the test suite is rerun, and further failures are again used to find and fix faults. This *sequential* mode of debugging was described in [10].

Failure clustering is a technique that can enable the *simultaneous* debugging of multiple faults, however one goal of our investigation is to determine whether the sequential mode of debugging — without clustering — is effective. Failure clustering can also be used to minimize the fault-localization interference among multiple faults. While existing work suggests that failure clustering can reduce interference [10, 12, 20], the introduction of another automated technique presents additional overhead in terms of computational and operational costs. These existing studies ex-

Table 1: The objects of our analysis, including for each: the number of faulty versions created for each quantity of faults, size of test suite, and number of test executions.

	Gzip	Replace	Space	Total
1 Fault	20	32	38	90
2 Faults	180	180	180	540
3 Faults	420	420	420	1260
4 Faults	550	550	550	1650
5 Faults	550	550	550	1650
6 Faults	550	550	550	1650
7 Faults	550	550	550	1650
8 Faults	550	550	550	1650
9 Faults	550	550	550	1650
10 Faults	550	550	550	1650
Total Number of Faulty Versions	4470	4470	4470	13,410
Number of Test Cases	214	5542	13,527	19,283
Total Number of Executions	956,580	24,772,740	60,465,690	86,195,010

amined the localizability of *specific* individual faults while, in practice, developers often aren't aware of the quantity or identity of the faults causing failures. In such situations, the localizability of *any* fault can enable an iterative debugging process that can lead to a fault-free program. While there are motivations for utilizing failure clustering, one question that we pose in this work is: *Are failure-clustering techniques necessary for effective fault localization when attempting to find a fault?*

3. EXPERIMENT

To understand the impact of the quantity of faults in a program on the effectiveness of CFL techniques, we conducted an experiment. In this section, we first describe the studied variables and measures. We then describe the objects for analysis. Next, we discuss the details of our experimental setup. Finally, we present our results of the experiment.

3.1 Variables and Measures

Our primary objective was to investigate the impact of the quantity of faults on the effectiveness of CFL techniques. Our experiment manipulated one independent variable: the quantity of faults in a program. We examined two dependent variables: (1) *Suspiciousness*: the average suspiciousness value assigned to the faulty instructions for each fault, and (2) *Expense*: the percentage of the program that must be examined to find the fault if examining the program in decreasing order of suspiciousness. The suspiciousness metric that we utilized is the Ochiai metric, defined by Equation 1. This metric was found by a study by Abreu and colleagues to be the most effective metric for CFL inferencing [1].

The expense metric measures the effectiveness of the CFL technique. This metric is computed by the following equation.

$$Expense = \frac{\text{rank of faulty statement}}{\text{number of executable statements}} * 100 \qquad (2)$$

An equivalent metric was originally presented by Renieris and Reiss [16] and used by many other researchers (e.g., [4, 8, 13]. When the fault was composed of multiple instructions, we calculated the *Expense* as the first faulty instruction found in the instructions sorted by decreasing suspiciousness.

3.2 Objects for Analysis

To determine whether CFL techniques are effective in the presence of multiple faults, we used three C-language programs which have been popular for CFL (e.g., [1, 4, 8–10, 13, 16]): Gzip (version 1.0.7), Replace, and Space. These programs vary in size: 768 LOC for Replace, 9251 LOC for Gzip, and 6445 LOC for Space. Each was obtained from the "Subject Infrastructure Repository" (SIR) [7] along with faults and test cases. In cases where the number of faults provided by the SIR was less than 20, we added additional faults by way of random mutation. We utilize the operators described in [14] for our mutant insertion. We attempted to create a representative set of mutants for each subject using randomized line selection and randomized mutant operator selection. As in prior work (e.g., [3, 8, 9, 13, 20]), faulty versions of the programs that exhibited no test failures were excluded from our experiment. When these omissions caused our program to have less than 20 faulty versions, additional mutations were made. To enable multiple faults to be simultaneously present in a program, we altered the reference version of the program in such a way that each fault could be activated or deactivated at compile-time.

3.3 Experimental Setup

In our experiment, we repeatedly applied a fault-localization technique to versions of our subject programs containing between 1 and 10 faults. For each, we compared its results against a reference version of the program that contained none of the injected faults. When the output of the faulty program differed from the output of the reference program, the test case was marked as a failure. When the outputs were equal, the test case was marked as a passing test case.

We captured the coverage of each test case using the instrumenter included in the Gnu C compiler (`gcc`) and its corresponding `gcov` utility. For each test case, the instructions that were executed and the pass/fail status were used as input to our fault-localization tool. We used a version of

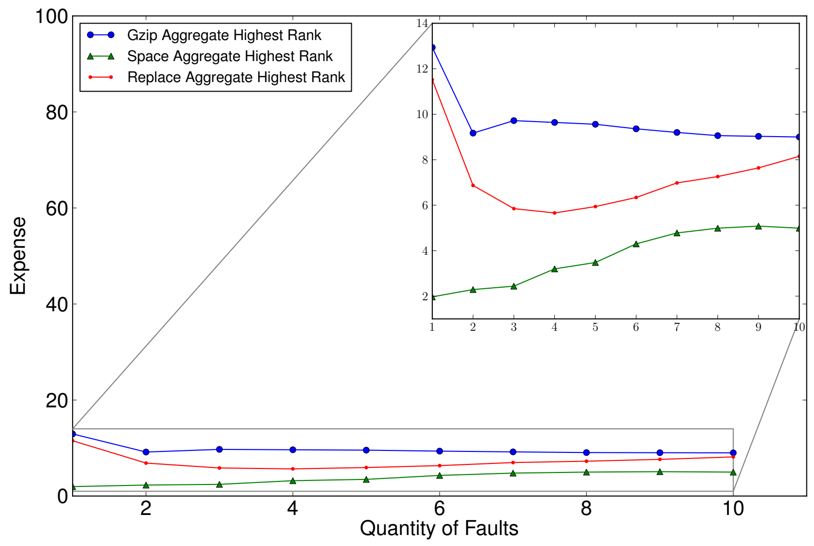


Figure 2: The aggregate, least expense, fault for all three subjects as the quantity of faults increases from one to ten. The inset plot presents a magnified view.

our TARANTULA fault-localization tool, utilizing the Ochiai suspiciousness metric, defined in Equation 1.

To explore the impact of the quantity of faults on the effectiveness of the CFL technique, we created versions of each of the programs containing 1, 2, . . . , 10 faults, chosen at random. The process for inclusion of faults proceeded by randomly choosing a single fault and iteratively adding another randomly-chosen fault until ten faults were reached. At each point, the test suite was executed and the CFL technique was utilized and evaluated. For example, we would evaluate a version containing only fault 7, followed by a version containing faults 7 and 14, followed by a version containing faults 7, 10, and 14, . . . , until ten faults were reached. By following such additive sequences of faults, we can capture, compute, and examine the progression of the *Expense* and *Suspiciousness* metrics as the introduced faults were included.

For each subject, for each quantity of faults, we generated up to 550 faulty versions. The exact number of faulty versions for each subject, for each quantity of faults is shown in Table 2.5. For example, for Gzip, we started with 20 single fault versions. By producing multiple combinations of two faults, we generated 180 two-fault versions (for example, a two-fault version may contain faults 4 and 17). We continued to generate n-fault versions by randomly choosing combinations of the individual faults. In all, we generated 13,410 faulty programs — across all three programs and all quantities of faults. Considering the number of test cases provided for each program (listed in Table 2.5) that were executed against all 13,410 programs, in total we executed over 86 million test cases (and thus captured that many coverage results).

3.4 Results

3.4.1 Expense Results

We represent the results concerning the effectiveness of the CFL technique utilizing the *Expense* dependent variable in Figure 3.2. In Figure 3.2, the horizontal axis represents the quantity of faults included in the program. The vertical axis represents the *Expense* to find the first fault, utilizing the order of examination provided by the fault-localization technique. The average expense for each of the object programs is represented by a separate plot line — the legend in the figure describes how to differentiate the plot lines. The plot is represented both at a scale of 0% to 100% — to represent the full range possible — and at a magnified scale in the inset plot. The results show that for all programs studied, the *Expense* varies very little with the quantity of faults. In fact, in two of the three subjects (Gzip and Replace), the most localizable fault was more localizable when ten faults are present than when a single fault is present.

We present the *Expense* that was computed for each of the fault quantities in each of the programs in Figures 3a, 3b, and 3c. In these figures, we varied the number of faults present in the program along the horizontal axis. The vertical axis represents the measured *Expense*. The plot points correspond with all n-fault versions. At the position in the plot where the quantity is 1, only a single plot point is drawn, representing the average *Expense* for all single-fault versions, because naturally, there is only one fault to localize. Similarly, when the quantity is 2, a new point is plotted — the lower plot point represents the average *Expense* for the lesser of the two faults' *Expense* values, and the upper plot point represents the average *Expense* for the greater of the two faults' *Expense* values. The points are connected with a line to enable the readers' ability to observe the trend in *Expense* as new faults are introduced. Because there can only be a second fault when there are two faults in the program, the 2-fault plot line has no value at the 1-fault horizontal axis point — hence each line makes its introduction at one point further along this axis. Note that in each of these figures, each plot point corresponds with multiple different faults — they are classified not by the individual fault, but by their position in the sorted *Expense* measure.

Figure 3a presents the *Expense* values for each fault across all faulty versions of the Gzip program. Likewise, Figure 3b presents the same results for Replace, and Figure 3c presents the results for Space. In each of these programs, the first found fault has a relatively low *Expense* (as also seen in Figure 3.2). The remainder of the obscured faults can be considerably higher in the ranked expense.

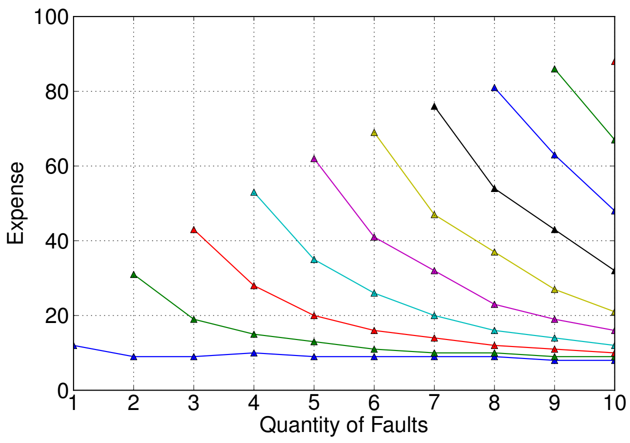
3.4.2 Suspiciousness Results

One concern that we had was that despite the fact that the *Expense* was relatively low for the most easily found fault regardless of the quantity of the faults, the overall, latent, suspiciousness may be comparatively high so as to make the localization of the faults difficult. To address this concern, we represent the *Suspiciousness* values assigned to each fault for each object program in Figures 4a, 4b, and 4c. In each of these figures, the horizontal axis represents the quantity of faults included in the program. The vertical axis represents the average *Suspiciousness* value. The points connected by the solid plot lines represent the average *Suspiciousness* value assigned to the n-th placed fault in the sorted list of instructions, from most suspicious to least. The top-most plot line represents the first fault to be found by the technique, and thus the most suspicious. The second plot line from the top represents the second fault to be found by the technique. Because there can only be a second fault when there are two faults in the program, this plot line has no value at the 1-fault horizontal axis point — hence each line makes its introduction at one point further along this axis. The average, overall suspiciousness value assigned to *all* instructions throughout the program (both faulty and non-faulty) is represented by the dotted line.

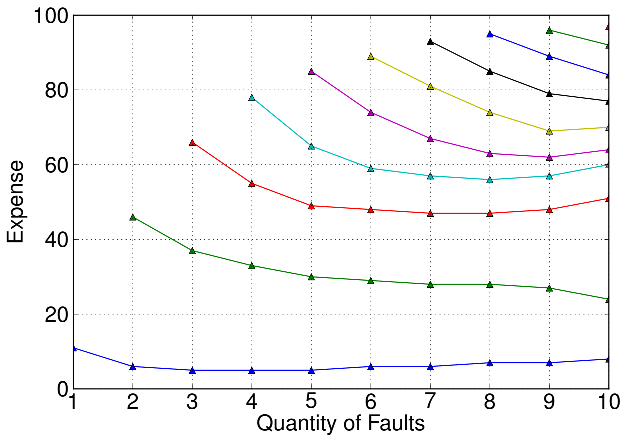
Note that in each of Figures 4a, 4b, and 4c, each plot point in a line corresponds with different *particular* faults — each faulty version may cause the rank to be sorted in a different order. For example, in a version that contains faults {1, 2, 3}, fault 2 may be ranked as the most suspicious fault; but, in a version that contains faults {2, 3, 4}, fault 4 may be ranked as the most suspicious fault.

Another point to note is that these suspiciousness values tend to be lower in value than some results presented in other works (e.g., [9]). In comparison with the original suspiciousness equation utilized by the TARANTULA technique, the Ochiai metric renders comparatively lower values. However, regardless of the suspiciousness metric used, the important property is the individual instructions' *relative* suspiciousness — all instructions will be sorted according to these values.

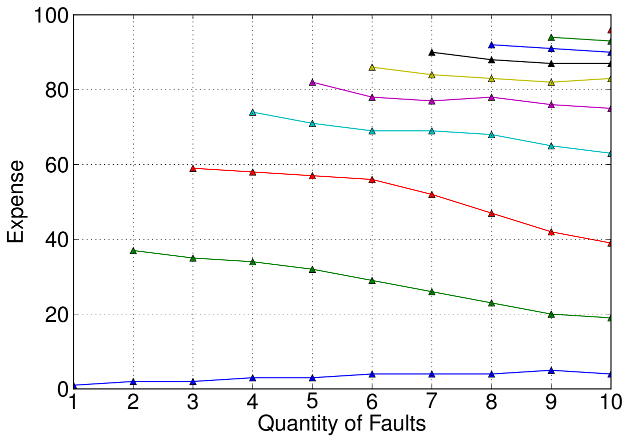
Figure 4a presents the suspiciousness results for Gzip. Notice that the overall *Suspiciousness* (dotted line) monotonically increases as the quantity of the faults increases. The *Suspiciousness* of each fault increases as the quantity of faults increases. The results show that even with ten faults, six of the faults are more suspicious than the average overall *Suspiciousness*.



(a) Expense results for Gzip.



(b) Expense results for Replace.



(c) Expense results for Space.

Figure 3: Aggregated expense for each fault, per subject. Lowest plot line represents least expense fault, and thus the first localized. Each of the other, higher lines represent next found faults.

Figure 4b presents the suspiciousness results for Replace. Like the results for Gzip in Figure 4a, the overall *Suspiciousness* monotonically increases as the quantity of faults increases. With five or more faults, two faults, on average, are more suspicious than the overall *Suspiciousness*.

Figure 4c presents the suspiciousness results for Space. Like the results for Gzip and Replace in Figures 4a and 4b, the overall *Suspiciousness* monotonically increases as the quantity of faults increases. With four or more faults, two faults, on average, are more suspicious than the overall *Suspiciousness*.

4. DISCUSSION AND ANALYSIS

To help interpret the results presented in Section 3.4, in this section we provide a discussion and analysis of them. We first examine how quantities of faults affect the effectiveness of CFL techniques. Next, we provide further analysis of how quantities of faults affect the suspiciousness of individual faults. Then, we identify and describe a phenomenon affecting the fault-localization effectiveness for individual faults in the presence of other faults. To further explore and demonstrate this phenomenon, we conducted a further, focused study and present its results. Finally, we discuss the implications of our results on the mode of debugging — sequential and parallel — to be assisted by CFL techniques.

4.1 Effects of Multiple Faults on CFL

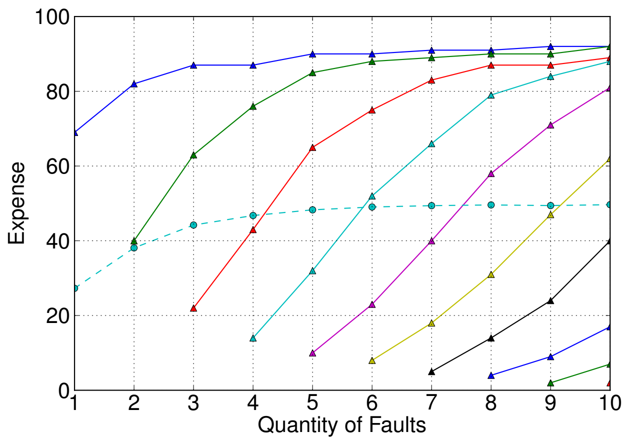
4.1.1 Effects of Multiple Faults on Expense

To determine the effects of multiple faults on a CFL technique, we examine the results presented in Figure 3.2, which presents the aggregated *Expense* for the most localized fault over all versions, at each quantity of faults, and for each subject. For the Space program with a single fault, our CFL technique was able to localize the fault with an *Expense* of just under 2% of the program, on average. The *Expense* slightly increases as the quantity of faults increases — our CFL technique provides an average *Expense* of 4.3% for the most localized fault when the program contains ten faults. In contrast, for Gzip containing a single fault, our CFL technique provided the most localized fault at 13%, and even less at 9% when it contained all ten faults. Similarly, for Replace containing a single fault, we obtained an *Expense* of 11.5%, and 8% with ten faults.

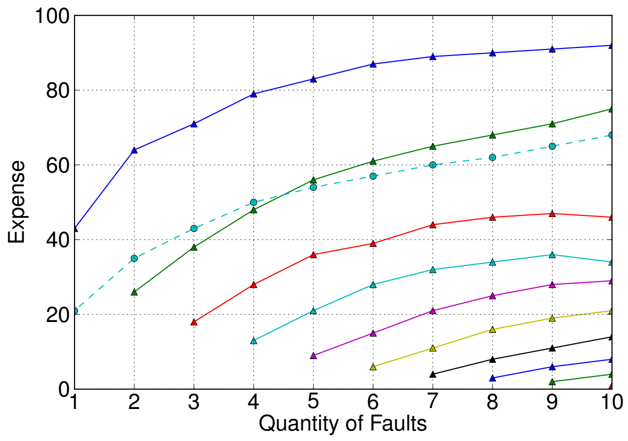
To summarize, the results presented in Figure 3.2 demonstrate that the *Expense* does not vary significantly with respect to the quantity of faults. This point contradicts that primary intuition that the effectiveness of CFL techniques suffers when multiple faults exist.

A further consideration is that in two subjects, Gzip and Replace, the CFL technique actually provided less average *Expense* when the program contained ten faults than the single-fault version. In other words, in two of the three subject programs, the CFL technique was more effective for the most localizable fault when it contained all ten faults, although the improvement was minor. We speculate that the decrease in *Expense* as the number of faults increases may be caused by the greater likelihood of the inclusion of a prominent, or more easily localizable, fault.

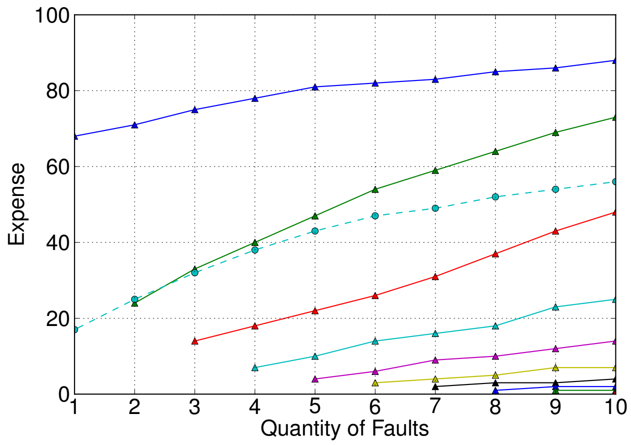
Figures 3a, 3b, and 3c provide a more complete view of the *Expense* measures for not only the localized fault, but also all other faults in the program. These results describe



(a) Suspiciousness results for Gzip.



(b) Suspiciousness results for Replace.



(c) Suspiciousness results for Space.

Figure 4: Average suspiciousness value for each fault per subject, organized by the place of the fault in the sorted list. The dotted line represents the average *Suspiciousness* of all instructions in the program.

the following procedure for a developer debugging a program with an unknown quantity, n, faults: (1) run the test suite and witness failures, (2) utilize a CFL technique to inform the localization of the fault, (3) find the most localizable fault (the bottom-most plot point), (4) fix that fault, (5) rerun the test suite and consequently find new failures, and (6) utilize the CFL technique to localize the next most localizable fault (the new bottom-most plot point moving left at the $n-1$ fault quantity). In other words, in interpreting these graphs and how they would affect developer expense, you should follow the *lowest plot line* to the *left* (not up) in the iterative find-and-fix debugging process. This is because after fixing the most localizable fault, a developer would rerun their test suite finding that a different fault now becomes localizable. And, while the faults that are localizable will be different in every iteration, in examining all three programs, it is evident that a single fault tends to dominate the CFL technique's ability to localize it. Notwithstanding the stability in the *Expense* for the most localizable fault, the other faults' *Expense* measures are also presented in our plots (the higher plot lines) to expose a number of properties such as the importance of the iterative localize-fix-rerun process and the impact of fault-localization interference.

We find these results to be reassuring because when the most localizable fault is found, fixed, and the test suite rerun against the "$n-1$"-th faulty version, the most localizable fault in this new version would also have a low *Expense*. This phenomenon can be inferred by examining Figures 3a, 3b, and 3c from right to left. As the quantity of faults decreases, the most localizable fault continues to be highly (at least relatively) localizable.

4.1.2 Effects of Multiple Faults on Suspiciousness

Zheng and colleagues [20] stated that the presence of multiple faults causes CFL techniques to not be able to distinguish useful faults predictors from non-useful ones. We investigated the effect of the quantity of faults on our predictor of faults, the *Suspiciousness* metric. Figures 4a, 4b, and 4c present these results. These figures plot the average *Suspiciousness* of all instructions throughout the program, and the *Suspiciousness* for each fault (categorized by their place in the sorted faults, from highest *Suspiciousness* to lowest).

For each program, the overall *Suspiciousness* does increase as the quantity of the faults increases. It increases by more than two times its single-fault value in Gzip and is almost tripled in Space and Replace. Figure 4c shows that when the Space program contained one fault, the average *Suspiciousness* is about 17%, whereas when it contained ten faults, the average *Suspiciousness* is 55%. However, we also observe that the most detectable fault's *Suspiciousness* (and indeed most every other fault) also increases.

We speculate that the overall *Suspiciousness* as well as the individual-fault *Suspiciousness* values increase with the number of faults because as more faults are included in the program, the number of test-case failures also increases. When the program fails on more test cases, each instruction is more likely to be executed by a greater number of failing test cases than passing ones.

The most suspicious fault when only one fault is active is around 68% for Space, 82% for Gzip and 42% for Replace. Yet these steadily increase until they plateau around 90% for all three subjects. Thus, as the overall *Suspiciousness* increases, at least one fault becomes so suspicious as to become easily localized. Figures 4a, 4b, and 4c provide a more complete view of the *Suspiciousness* measures for not only the localized fault, but also all other faults in the program. In interpreting these graphs and how they would affect developer suspiciousness, you should follow the *highest plot line* to the *left* (not down) in the iterative find-and-fix debugging process. This is because after fixing the most localizable fault, a developer would rerun their test suite finding that a different fault now becomes the most suspicious. Notwithstanding the increase in the *Suspiciousness* for the most localizable fault, the other faults' *Suspiciousness* measures are also presented in our plots (the lower plot lines) to expose a number of properties such as the importance of the iterative localize-fix-rerun process and the impact of fault-localization interference.

However, in all three subjects, a significant percentage of the faults have *Suspiciousness* scores that are less than the average line of code. This means that they would be indistinguishable from an average non-faulty instruction to a developer using the CFL technique. However, notwithstanding this phenomenon, the CFL technique is able to localize at least a single fault as far more suspicious than the overall, latent *Suspiciousness* of the program.

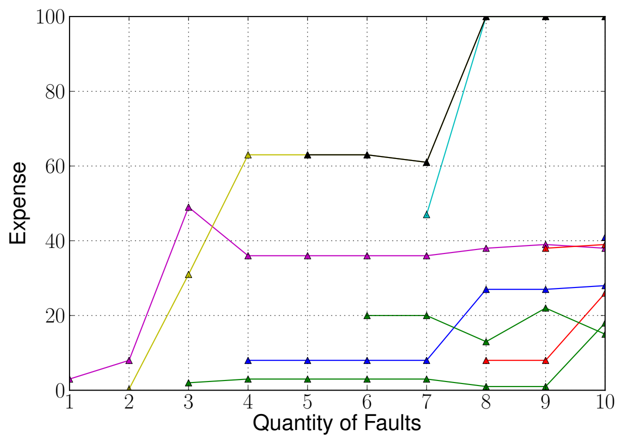


Figure 5: The *Expense* for individual faults following a particular sequence of 1–10 faults in Space. Notice that the expense for a particular fault can drastically change as a new fault is introduced or removed.

4.2 Prevalence of Fault-Localization Interference

The results presented thus far have been aggregated across many versions — up to 550 versions at each quantity of faults for each subject. To give the reader a better understanding of how the fault-localization effectiveness changes for a particular sequence of non-aggregated faults as the quantity of faults is altered, we present a small, focused study. Figure 4.2 presents the *Expense* for one particular sequence of faults that was examined in our experiment for the Space program. For the sake of replication of our results, the sequence of faults, in the order of introduction, is ⟨17, 20, 14, 9, 21, 34, 16, 15, 36, 26⟩. With only a single fault—fault

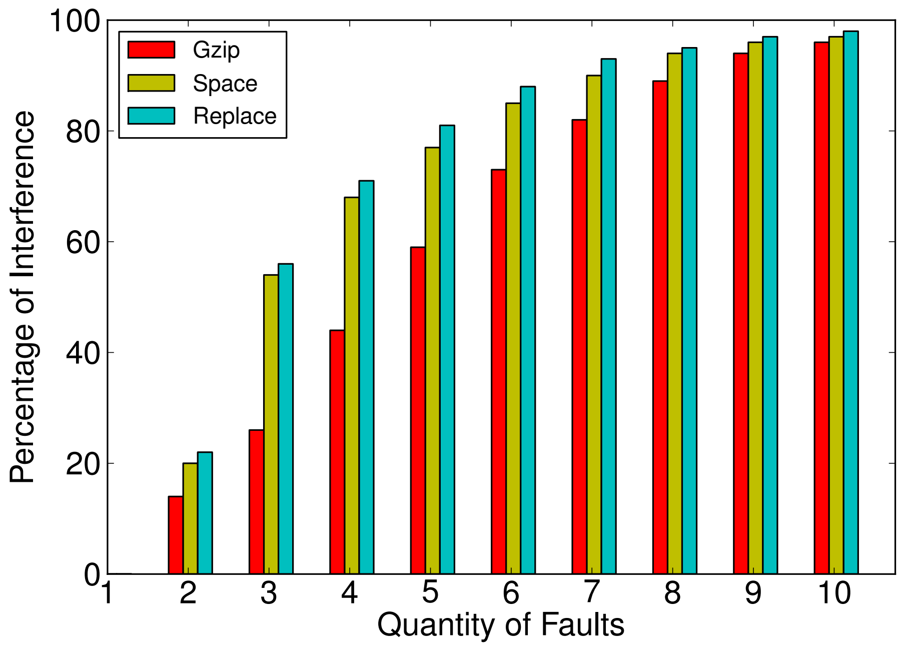


Figure 6: The amount of fault-localization interference that occurs at each quantity of faults. As the number of faults increases, the amount of fault-localization interference increases drastically.

17—the *Expense* to find that fault is only 3%. Upon introduction of the next fault—fault 20—fault 17's *Expense* jumps to 8% and fault 20's *Expense* is 0.3%. When the third fault is introduced—fault 14—the two existing faults' *Expense*s jump drastically to 31% and 49%.

In these cases, the additional faults acted to interfere with the localizability of the previous faults — that is, the presence of the fault reduced the ability of the CFL technique to effectively localize the other faults. Such changes in the effectiveness of the CFL technique to localize a particular fault indicate instances of *fault-localization interference* (recall its definition in Section 2.4).

One of the most pronounced examples of fault-localization interference in this detailed study is seen when the eighth fault—fault 15—is introduced. Once fault 15 is introduced, three other faults have their *Expense* measure jump to 100%. In these cases, these three obscured faults are no longer executed by any failing test cases.

We encountered many such examples while examining each of the three programs and the many sequences from one to ten faults. All of the three programs presented fault-localization interference. We present the degree of interference that we found in each of the three subject programs in Figure 4.2. The quantity of faults is represented on the horizontal axis and the percentage of the versions that exhibited fault-localization interference is represented on the vertical axis. For the purposes of this chart, we refer to a version that exhibits fault-localization interference as one where the *Expense* of at least one of its faults increased by at least an order of magnitude over its single-fault *Expense*. However as noted in Section 2.4 any ineffectiveness is technically interference, we use an order of magnitude as a degree of ineffectiveness developers would consider significant. Figure 4.2 demonstrates that as the quantity of faults was increased the degree of interference increases drastically — at ten faults, all three subjects exhibited over 90% of versions containing interference.

These results demonstrate the interference among faults, along with their inability to be localized independently with CFL techniques is prevalent. This prevalence implies that clustering of failures for the purposes of applying CFL techniques on individual faults is likely to encounter some difficulties in at least some of the faults — the faults that have been obscured by more dominant, masking faults. In this way, regardless of the use of failure clustering a degree of iterative debugging is nearly inevitable. These results are also confirmed by Jones and colleagues in [10] where multiple iterations were needed to localize all faults, despite the failure clustering and parallelization that was performed. They found that "some faults prevent others from being active."

These results also help us understand what to expect during regression testing. Because fault-localization interference is so prevalent it will be common that after correctly fixing a fault and rerunning a test suite, a developer will discover new failures. Such situations may lead to confusion about whether a fault was fixed correctly, or in attempting to fix it, more faults were introduced. Our results also increase our understanding of time estimation. Estimating time that it will take to fix features will need to account obscured faults adding another layer of complexity but enabling a more accurate estimate.

4.3 To Cluster or Not to Cluster?

Given the prevalence of fault-localization interference and obscuring faults, and the relative effectiveness of CFL techniques regardless of the number of faults in a program, the practice of localizing and providing automated assistance to the most detectable fault appears to be a practical option. The mode of debugging that would be imagined would entail the developer utilizing the CFL technique to provide assistance in finding faults. The program and the fault-localization results would be examined to find the most prevalent fault. At this point, the natural inclination of the developer would likely be to fix that fault and re-execute the test suite to confirm that she was successful in fixing the fault. If she was successful in fixing the fault, additional test cases would fail, and the CFL technique would again give a similarly effective prediction of the location of the most prevalent fault.

These results, however, do not invalidate the use of failure clustering. Indeed, failure clustering can be an effective precursor to CFL techniques, as evidenced by studies in References [10, 20]. Such clustering techniques can minimize, although not eliminate, some fault-localization interference. The effectiveness of the subsequent CFL technique may benefit from less "noise" in localizing the most prevalent fault for each cluster, although, from our results, the quantity of faults has negligible effects on the effects of fault localization. One primary motivation for failure clustering is to enable the debugging of multiple faults by multiple developers, in parallel. Another motivation can be to aid in find a particular fault, given a specific failure. In these cases—debugging in parallel and finding similar failures to a specific failure—failure clustering provides a practical benefit.

Another situation benefiting from clustering is when developers wish to fix a *specific* fault. This work defines success as localizing *any* fault, but there are circumstances where developers need to overcome a specific failure, and clustering would be valuable to accomplish that. This is because the most localizable fault may not be the one that developers want to find, however this scenario requires an investigation.

However, the result of our studies here is that CFL techniques can be effectively utilized, without the use of failure clustering. Failure clustering adds a level of computational cost and an additional imposition of tool support and development-practice changes, and while its use can be beneficial, it is not strictly necessary unless the goal is to enable parallel debugging by multiple developers or localization of one specific failure's fault.

5. THREATS TO VALIDITY

Some of the difficulties in creating external validity of this work pertain to specific difficulties in generalization. We only evaluated our subjects from one to ten faults, and in so doing, we cannot make any claims about how these results will generalize above ten faults. However, Figure 3.2 displays trends which are fairly smooth and consistent. It seems unlikely the behavior will drastically change in the presence of more faults. Another concern with generalizability is that we only tested three subjects. With the large diversity of software that exists in both size and complexity it would be very difficult to test a large enough sample to ensure that our results would hold across all software. However two of the three programs we used in our experiments are real-world software. We believe this increases likelihood of generalizability across all software. Because more than 13,000 versions were created which executed over 86 million test cases, we expect the results from our study to be a representative set for programs of similar size.

Another issue we faced was the need to insert mutants into our code. The nature of this study required more faults than were delivered by SIR [7] which imposed a need for mutant creation. While it is true that our faults were not "real" faults, Offutt and colleagues' study [14] demonstrates that if mutants are created properly, they can be both "effective and efficient" in their ability to test a program. To minimize this concern, all of our mutants were made in accordance with Offutt and colleagues' suggestions.

A difficulty in creating construct validity is our assumption of how a user would use the information our experiments provide. We assume a user will examine the rankings and select the lowest ranked (i.e., most suspicious) instruction and determine whether it is faulty. If that instruction is not faulty, we expect them to then examine the next lowest rank, and so forth. If this process is not followed, our claim of effectiveness could be inaccurate. We cannot know how each developer will use such techniques unless this issue is investigated through an observational study with a significant quantity of developers with varying levels of expertise.

6. CONCLUSIONS

In this paper, we presented an experiment to examine how CFL effectiveness is affected by fault quantity. We provided evidence that is contrary to a commonly held belief that CFL techniques cannot perform effectively in the presence of multiple faults. On one hand, we found the beliefs and intuitions of faults creating "noise" or interference that inhibits the effectiveness of coverage-based fault localization to be well founded — this interference exists and is prevalent. On the other hand, we also found that this interference did not adversely affect the localizability of at least one, most prominent, fault. Our results show that CFL techniques continue to be effective in the presence of multiple faults in spite of interference, and in many cases increase in effectiveness. CFL's effectiveness derives from at least one fault becoming so suspicious that CFL techniques are able to identify it. Thus, CFL tools can be used regardless of fault quantity and still be expected to perform well.

In regards to failure clustering, our results show that unless the goal is the *simultaneous* debugging of multiple faults, or the localization of a *specific* fault, it is not necessary as a precursive step to CFL. While failure clustering can aid in the reduction of interference when attempting to locate a *specific* fault, developers often aren't aware of the quantity, or identity of the faults in their programs. Thus, the ability to localize *any* fault is beneficial. We presented evidence that CFL effectively localizes at least one fault regardless of fault quantity, eliminating the requirement for clustering prior to utilizing CFL techniques.

Finally, fault-localization interference was discovered to be a very real concern for CFL in situations where multiple faults existed. We defined fault-localization interference and gave examples of it in real-world software. Fault-localization interference's influence is significant because as the quantity of faults increases, the chances that an obscuring fault is present becomes increasingly likely. The presence of an interfering fault creates difficulties in the localizing of other faults that are being obscured. Such interference also makes identifying how many faults exist nearly impossible with current methods, as they can only be observed after the obscuring fault is removed.

Although our studies provide practical and useful insights into the applicability of coverage-based fault localization techniques, more experimentation must be conducted to verify the results in general. Specifically, we are planning experiments on larger programs with even more faults. We would also like to further investigate the nature of the influence and interference among faults. Such investigations of fault interference have the potential to impact and inform new failure-clustering and debugging-time-estimation research.

7. REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference, Practice and Research Techniques*, Windsor, UK, September 2007.

[2] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical fault localization for dynamic web applications. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 265–274, New York, NY, USA, 2010. ACM.

[3] P. Arumuga Nainar and B. Liblit. Adaptive bug isolation. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 255–264, New York, NY, USA, 2010. ACM.

[4] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the International Conference on Software Engineering*, pages 342–351, St. Louis, Missouri, May 2005.

[5] V. Debroy and W. E. Wong. Insights on fault interference for programs with multiple bugs. In *Proceedings of the 2009 20th International Symposium on Software Reliability Engineering*, ISSRE '09, pages 165–174, Washington, DC, USA, 2009. IEEE Computer Society.

[6] T. Denmat, M. Ducassé, and O. Ridoux. Data mining and cross-checking of execution traces. A re-interpretation of Jones, Harrold and Stasko test information visualization (Long version). Research Report RR-5661, INRIA, August 2005. Also Publication Interne IRISA PI-1743.

[7] H. Do, S. Elbaum, and G. Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 60–70, Washington, DC, USA, 2004. IEEE Computer Society.

[8] J. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the International Conference on Automated Software Engineering*, pages 273–282, November 2005.

[9] J. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering*, pages 467–477, Orlando, Florida, May 2002.

[10] J. A. Jones, J. F. Bowring, and M. J. Harrold. Debugging in parallel. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 16–26, New York, NY, USA, 2007. ACM.

[11] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 2005.

[12] C. Liu and J. Han. Failure proximity: A fault localization-based approach. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, pages 286–295, November 2006.

[13] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In *Proceedings of 10th European Software Engineering Conference and 13th Foundations on Software Engineering*, pages 286–295, September 2005.

[14] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5:99–118, April 1996.

[15] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the International Conference on Software Engineering*, pages 465–474, May 2003.

[16] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the International Conference on Automated Software Engineering*, pages 30–39, Montreal, Quebec, October 2003.

[17] M. Srivastav, Y. Singh, C. Gupta, and D. Chauhan. Complexity estimation approach for debugging in parallel. In *Computer Research and Development, 2010 Second International Conference on*, pages 223 –227, May 2010.

[18] I. Vessey. Expertise in debugging computer programs. *International Journal of Man-Machine Studies: A process analysis*, 23(5):459–494, 1985.

[19] C. Yilmaz, A. Paradkar, and C. Williams. Time will tell: fault localization using time spectra. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 81–90, New York, NY, USA, 2008. ACM.

[20] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning*, ICML '06, pages 1105–1112, New York, NY, USA, 2006. ACM.