An Empirical Study of the Effects of Test-Suite Reduction on Fault Localization

Yanbing Yu
yyu@cc.gatech.edu

James A. Jones
jjones@cc.gatech.edu

Mary Jean Harrold
harrold@cc.gatech.edu

College of Computing
Georgia Institute of Technology
Atlanta, GA, U.S.A.

ABSTRACT

Fault-localization techniques that utilize information about all test cases in a test suite have been presented. These techniques use various approaches to identify the likely faulty part(s) of a program, based on information about the execution of the program with the test suite. Researchers have begun to investigate the impact that the composition of the test suite has on the effectiveness of these fault-localization techniques. In this paper, we present the first experiment on one aspect of test-suite composition—test-suite reduction. Our experiment studies the impact of the test-suite reduction on the effectiveness of fault-localization techniques. In our experiment, we apply 10 test-suite reduction strategies to test suites for eight subject programs. We then measure the differences between the effectiveness of four existing fault-localization techniques on the unreduced and reduced test suites. We also measure the reduction in test-suite size of the 10 test-suite reduction strategies. Our experiment shows that fault-localization effectiveness varies depending on the test-suite reduction strategy used, and it demonstrates the trade-offs between test-suite reduction and fault-localization effectiveness.

Categories and Subject Descriptors

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

General Terms

Experimentation, Reliability

Keywords

Fault localization, test-suite reduction, empirical study

1. INTRODUCTION

Debugging software after it has failed is an expensive part of the software-development process. To determine the location of a fault that has caused a failure, software developers often use manual and tedious methods. With limited diagnostic techniques, such as inserting print statements into the code and using a symbolic debugger, finding the fault, or *fault localization*, can often consume a significant amount of time and resources.

To address the expense of fault localization, researchers have proposed automated fault-localization techniques. These techniques typically use dynamic information obtained from executing the program to direct developer attention to likely faulty locations, and thus, reduce the expense of the search. Many of these fault-localization techniques use information about the execution of a test suite of test cases for this identification (e.g., [1, 12, 13, 14]). Along with the pass/fail results of the execution of the test suite, these techniques are based on some type of coverage information about the test suite's executions. We call these *coverage-based* fault-localization techniques.[1] Studies show that these techniques can be helpful in reducing the fault-localization expense by reducing the percentage of the program that must be inspected to find the fault (e.g., [6, 11, 16]).

One issue related to these coverage-based fault-localization techniques is the effect that the composition of the test suite has on the fault localization. To date, several researchers have begun to investigate the way in which the composition of the test suite impacts the effectiveness of fault-localization techniques. Two recent papers report studies of the effects of increasing the size of the test suite used for fault localization. Abreu and colleagues [1] randomly select test suites of varying numbers of passed and failed test cases. For their fault-localization technique and the subject programs and test suites, they found that including more than six failed test cases or more than twenty passed test cases produces minimal effects on the effectiveness of the fault-localization technique. However, they give no general approach for finding these bounds on the passed and failed test cases. Baudry and colleagues [4] increase the size of the test suite by selectively adding test cases. They define a *dynamic basic block* as a set of statements that is covered by the same test cases. Using a test-case generator, they propose keeping test cases that increase the number of dynamic basic blocks and discarding test cases that do not. They found that increasing the number of dynamic basic blocks increased the effectiveness of the fault-localization technique. Neither of these techniques, however, considers how removing test cases from the test suite can affect the fault localization.

ICSE'08, May 10–18, 2008, Leipzig, Germany.
Copyright 2008 ACM 978-1-60558-079-1/08/05 ...$5.00.

[1]These techniques have also been called *spectra-based* because each execution generates a spectrum for the execution.

One recent paper does consider reducing the test suite. Hao and colleagues [7] posit that test-case similarity or redundancy results in a loss of fault-localization effectiveness. They performed an empirical study to show that injected redundancy can impair a fault-localization technique's effectiveness. Their results suggest that reduction could improve effectiveness. Our preliminary results contradict this result as we found that additional redundancy does not, in general, reduce the effectiveness of fault-localization techniques.

To address these issues of test-suite composition, we have begun a project that will consider different methods of composing test suites. In this paper, we present the results of our first experiment on test-suite composition, in which we investigate the effects that test-suite reduction strategies have on the effectiveness of fault-localization techniques. In the experiment, we used 10 test-suite reduction strategies and four existing fault-localization techniques, along with a set of programs, containing single and multiple faults, and a large number of test suites. Our experiment shows the trade-offs that exist between test-suite reduction and fault-localization effectiveness. Our experiment also shows that, in general, existing test-suite reduction strategies reduce the effectiveness of fault-localization techniques. In the paper, we also propose a new test-suite reduction strategy and show that, for our subject programs and test suites, it causes negligible impact on the effectiveness of the four fault-localization techniques.

The main contributions of this paper are:

- The first controlled experiment (to our knowledge) that evaluates the effectiveness of test-suite reduction on fault-localization effectiveness, using both single- and multiple-fault programs.
- A description of new test-suite reduction strategies, along with empirical evidence that they maintain the fault-localization effectiveness on the reduced test suites.
- Evidence that contradicts an existing report claiming that redundancy in test suites is a source of error for fault localization.

2. TECHNIQUES

In this section, we present the relevant test-suite reduction strategies and the fault-localization techniques that we used for our experiment.

2.1 Test-Suite Reduction

2.1.1 Test-Suite Reduction Overview

Test-suite reduction techniques identify a subset of a test suite that maintains some characteristic of the test suite, such as coverage. Those entities whose coverage is measured (i.e., *coverage entities*) are called *test-case requirements.* For example, test-case requirements could be a program's statements, branches, or system requirements. Given this set of test-case requirements, the test-suite reduction problem can be stated as follows [8]:

Given: Test suite TS, a set of test-case requirements $r_1, r_2, \ldots, r_n$ that must be satisfied to provide the desired test coverage of the program, and subsets of TS, $T_1, T_2, \ldots, T_n$, one associated with each of the r_is such that any one of the test cases t_j belonging to T_i can be used to test r_i.

Problem: Find a representative set of test cases from TS that satisfies all of the r_is.

A test suite that satisfies all r_is must contain at least one test case from each T_i. Such a set is called a hitting set of the T_i. Maximum reduction is achieved with the minimum-cardinality hitting set of the T_is. Because the problem of finding the minimum-cardinality hitting set is intractable, test-suite reduction techniques must approximate the minimum cardinality. A number of algorithms have been developed for use in test-suite reduction (e.g., [5, 8, 9, 15]).

2.1.2 Test-Suite Reduction Strategies

The test-suite reduction strategies that we use for our experiment have two dimensions: (1) the test-case requirements used for the reduction and (2) the test set being considered in the reduction.

For the first dimension of our test-suite reduction strategies, we consider two test-case requirements on which to apply the reduction: statement-based and vector-based. *Statement-based* reduction (abbreviated as S), an often-used test-suite reduction strategy (e.g., [8]), has as its goal to produce a reduced test suite that executes the same set of statements as the unreduced test suite. Thus, the test-case requirements for this strategy are the statements in the program. To illustrate, consider the program and test suite shown in Figure 1. Program *mid()* inputs three integers and outputs the median value of the three integers. To the right of the code is information about a test suite of eight test cases: inputs are shown at the top of each column, coverage is shown by the black dots, and pass/fail status is shown at the bottom of the columns. To the right of the test suite are several columns that relate to fault localization; these columns will be described in Section 2.2. For statement-based reduction, the test-case requirements are statements $s1, s2, ..., s13$, and the test suite shown covers all statements except $s12$. Statement-based reduction could result in {t1, t2, t3, t4} because this subset of the test suite also covers all statements in the program except $s12$ (i.e., it satisfies the same test-case requirements). In this case, t5, t6, t7, and t8 provide no additional statement coverage over {t1, t2, t3, t4}. More than one reduced test suite can satisfy the same test-case requirements as the unreduced test suite. For our example, test suites {t1, t2, t3, t4, t5}, {t2, t3, t4, t7}, and {t2, t3, t4, t5, t7} are also reduced test suites that satisfy the same test-case requirements as the unreduced test suite.

Vector-based reduction (abbreviated as V), our new test-suite reduction strategy, has as its goal to produce a reduced test suite that executes the same set of statement vectors as the unreduced test suite. A *statement vector* is the set of statements executed by one test case. To illustrate, consider again the program and test suite shown in Figure 1. For vector-based reduction, the test-case requirements are the statement vectors in the program. Test cases t1, t7, and t8 each executes statement vector $<s1, s2, s3, s4, s6, s7, s13>$. Thus, to maintain vector coverage, one of these test cases must be in any reduced test suite. Vector-based reduction could result in test suite {t1, t2, t3, t4, t5}. In this case, t6, t7, and t8 provide no additional vector coverage over {t1, t2, t3, t4, t5}. For the example, there are also other reduced test suites, such as {t2, t3, t4, t5, t7} and {t2, t4, t5, t6, t8}, that satisfy the same test-case requirements as the unreduced test suite.

For the second dimension of our test-suite reduction strate-

	Test Cases								Tarantula			SBI		Jaccard		Ochiai	
	t1	t2	t3	t4	t5	t6	t7	t8	suspiciousness$_T$	confidence	rank	suspiciousness$_S$	rank	suspiciousness$_J$	rank	suspiciousness$_O$	rank
`mid() {` `int x,y,z,m;`	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	7,5,4	2,1,3	4,3,5									
`1: read("Enter 3 numbers:",x,y,z);`	●	●	●	●	●	●	●	●	0.5	1.0	7	0.25	7	0.25	7	0.5	7
`2: m = z;`	●	●	●	●	●	●	●	●	0.5	1.0	7	0.25	7	0.25	7	0.5	7
`3: if (y<z)`	●	●	●	●	●	●	●	●	0.5	1.0	7	0.25	7	0.25	7	0.5	7
`4:    if (x<y)`	●	●			●		●	●	0.67	1.0	3	0.4	3	0.4	3	0.63	3
`5:       m = y;`		●							0.0	0.17	12	0.0	13	0.0	13	0.0	13
`6:    else if (x<z)`	●				●		●	●	0.75	1.0	2	0.5	2	0.5	2	0.71	2
`7:       m = y;  // *** bug ***`	●						●	●	0.86	1.0	1	0.67	1	0.67	1	0.82	1
`8: else`			●	●		●			0.0	0.5	9	0.0	13	0.0	13	0.0	13
`9:    if (x>y)`			●	●		●			0.0	0.5	9	0.0	13	0.0	13	0.0	13
`10:      m = y;`			●			●			0.0	0.33	10	0.0	13	0.0	13	0.0	13
`11:   else if (x>z)`				●					0.0	0.17	12	0.0	13	0.0	13	0.0	13
`12:      m = x;`									0.0	0.0	13	0.0	13	0.0	13	0.0	13
`13: print("Middle number is:",m);`	●	●	●	●	●	●	●	●	0.5	1.0	7	0.25	7	0.25	7	0.5	7
`}` Pass/Fail Status	P	P	P	P	P	P	F	F									

Figure 1: Example program, information about its test suite, and its rank results for the four fault-localization techniques.

gies, we consider the subset of the test cases in the test suite on which the reduction is performed. We apply the reduction to five types of test sets in the test suite: (1) All, (2) Passed, (3) Failed, (4) Passed and Failed, and (5) All with preference for failed. The first and most traditional test set consists of all test cases in the test suite, or *All*. For this test set, all test cases in the test suite are considered equally in the reduction. The second test set consists of all passed test cases in the test suite, or *Passed*. For this test set, the reduction is performed only on the passed test cases, with no reduction of the failed test cases. The third test set consists of the failed test cases, or *Failed*. For this test set, the reduction is performed on the failed test cases, with no reduction on the passed test cases. The fourth test set consists of the set of passed and the set of failed test cases, or *Passed and Failed*. For this test set, each group of test cases—passed and failed—is reduced in isolation and then the reduced sets are combined to form the reduced test suite. The fifth test set consists of the entire test suite with preference in reduction given to failed test cases, or *All with preference for failed*. For this test set, the reduction is performed like the *All* approach except that whenever a passed test case and a failed test case are equal candidates for keeping in the reduced test suite, the failed test case is selected.

Combining the two dimensions—the test-case requirements and the test set being considered—results in 10 test-suite reduction strategies. The abbreviated expression and brief description for each strategy is shown in the following.

***SA*:** statement-based reduction on all test cases;
***SP*:** statement-based reduction only on passed test cases;
***SF*:** statement-based reduction only on failed test cases;
***SPF*:** statement-based reduction on both passed and failed test cases in isolation;
***SR*:** statement-based reduction on all test cases with preference for failed test cases;
***VA*:** vector-based reduction on all test cases;
***VP*:** vector-based reduction only on passed test cases;
***VF*:** vector-based reduction only on failed test cases;
***VPF*:** vector-based reduction on both passed and failed test cases in isolation;
***VR*:** vector-based reduction on all test cases with preference for failed test cases;

To illustrate the 10 strategies, again consider the program and test suite in Figure 1. Table 1 shows, for each test-suite reduction strategy, one possible reduction result.

Table 1: Test-suite Reduction Results on `mid()`.

Strategy	Reduced Test Suite.
SA	{t1, t2, t3, t4}
SP	{t1, t2, t3, t4, t7, t8}
SF	{t1, t2, t3, t4, t5, t6, t7}
SPF	{t1, t2, t3, t4, t7}
SR	{t2, t3, t4, t7}
VA	{t1, t2, t3, t4, t5}
VP	{t1, t2, t3, t4, t5, t7, t8}
VF	{t1, t2, t3, t4, t5, t6, t7}
VPF	{t1, t2, t3, t4, t5, t7}
VR	{t2, t3, t4, t5, t7}

2.2 Fault-Localization Techniques

Researchers have proposed a number of techniques for providing automated assistance to developers in searching for faults. Many of these techniques use coverage information about a test suite to identify the likely faulty part(s) of the program (e.g., [2, 6, 12, 13, 14, 16]). This section presents an overview of four of these coverage-based fault-localization techniques that we used in our experiment.

2.2.1 Tarantula

In prior work [11, 12], we present one coverage-based technique called TARANTULA. TARANTULA assigns two metrics to each coverage entity in the program—*suspiciousness* and *confidence*—based on the numbers of passed and failed test cases in a test suite that executed that coverage entity.[2] TARANTULA can be applied to various coverage entities, such as branches, statements, and invariants. However, in this discussion, we use statements as the coverage entities. The intuition behind TARANTULA's metric for fault localization is that statements in a program that are primarily executed

[2]We use subscripts to differentiate the suspiciousness values of the four fault-localization techniques we studied.

by failed test cases are more likely to be faulty than those that are primarily executed by passed test cases. The suspiciousness of a statement s is computed by

$$\mathit{suspiciousness}_T(s) = \frac{\%\text{failed}(s)}{\%\text{failed}(s) + \%\text{passed}(s)} \quad (1)$$

In Equation 1, %failed(s) is a function that returns, as a percentage, the ratio of the number of failed test cases that executed s to the number of failed test cases in the test suite. %passed(s), likewise, is a function that returns, as a percentage, the ratio of the number of passed test cases that executed s to the number of passed test cases in the test suite. If the denominator in this ratio is zero, TARANTULA assigns zero suspiciousness. The suspiciousness can range from 0, indicating that a statement is not suspicious, to 1, indicating that a statement that is highly suspicious.

The confidence metric is meant to measure the degree of confidence in the given suspiciousness. The TARANTULA techniques assigns a greater confidence to statements that are covered by more test cases. The confidence of a statement is computed by

$$\mathit{confidence}(s) = \mathit{max}(\%\text{failed}(s), \%\text{passed}(s)) \quad (2)$$

Using suspiciousness and confidence, the TARANTULA technique sorts the coverage entities of the program under test to provide a *rank* for each statement. The technique first sorts on decreasing suspiciousness, and breaks ties by next sorting on decreasing confidence. For statements tied with the same suspiciousness and confidence, the technique assigns them a rank as the sum of the number of the tied statements and the number of statements ranked before them.[3] The set of statements that have the highest rank should be considered first by the developer when searching for the fault. If, after examining these statements, the fault is not found, the developer can examine the remaining statements in order of decreasing rank.

To illustrate the TARANTULA technique, consider the example in Figure 1. The TARANTULA suspiciousness, confidence, and resulting rank for each statement are shown in the columns to the right of the test suite. The program contains a fault on line 7—this line should read "`m = x;`". According the rank induced by the suspiciousness and confidence, the developer would be directed first to focus attention on line 7 as it is the statement with the highest rank.

2.2.2 *Statistical Bug Isolation (SBI)*

Liblit and colleagues [13] proposed a similar technique, called STATISTICAL BUG ISOLATION (SBI)[4] for computing the suspiciousness of a predicate P, which they call *Failure*. With the assumption that the probability of P being true implies failure, they compute the *Failure* of P by

$$\mathit{Failure}(P) = \frac{\mathit{failed}(P)}{\mathit{passed}(P) + \mathit{failed}(P)} \quad (3)$$

where passed(P) is the number of passed test cases in which P is observed to be true and failed(P) is the number of failed test cases in which P is observed to be true.

[3]This rank computation, presented by Renieris and Reiss [16], has been used for evaluation and comparison of fault-localization techniques.

[4]In recent work, the project has been renamed Collaborative Bug Isolation (CBI).

To facilitate comparison among TARANTULA, SBI, and the other fault-localization techniques, we adapted Equation 3 to compute the suspiciousness of a statement s or *Failure*(s) by considering the predicate to be whether s is executed. In the adapted equation, passed(s) is the number of passed test cases that executed s and failed(s) is the number of failed test cases that executed s. We represent the *Failure* as $\mathit{suspiciousness}_S$.

$$\mathit{suspiciousness}_S(s) = \frac{\mathit{failed}(s)}{\mathit{passed}(s) + \mathit{failed}(s)} \quad (4)$$

Given the $\mathit{suspiciousness}_S$ values computed for each statement, the rank is computed as described in Section 2.2.1. SBI also uses other metrics, *Context* and *Increase*. However, in this application of the technique, statement-coverage predicates without selective sampling of predicate observations, these metrics do not influence the ranking.

To illustrate the SBI technique, again consider the example in Figure 1. The SBI suspiciousness and ranks are shown in columns to the right of the TARANTULA results.

2.2.3 *Jaccard*

In recent work, Abreu and colleagues [1] compare TARANTULA's suspiciousness with the JACCARD metric in terms of its diagnostic accuracy. The JACCARD index, also known as the JACCARD similarity coefficient, is a metric used for comparing the similarity and diversity of sample sets.

The JACCARD equation used in Reference [4] can be represented as

$$\mathit{suspiciousness}_J(s) = \frac{\mathit{failed}(s)}{\mathit{totalfailed} + \mathit{passed}(s)} \quad (5)$$

where passed(s) and failed(s) are the same as in Equation 4, and *totalfailed* is the number of failed test cases in the test suite. Given the $\mathit{suspiciousness}_J$ values computed for each statement, the rank is computed as described in Section 2.2.1. Figure 1 shows the JACCARD suspiciousness and ranks.

2.2.4 *Ochiai*

In recent work, Abreu and colleagues [1] also compare TARANTULA's suspiciousness with the OCHIAI coefficient, which originated in the molecular biology domain. The equation for OCHIAI can be represented as

$$\mathit{suspiciousness}_O(s) = \frac{\mathit{failed}(s)}{\sqrt{\mathit{totalfailed} * (\mathit{failed}(s) + \mathit{passed}(s))}} \quad (6)$$

where passed(s), failed(s), and totalfailed are the same as in Equation 5. Given the $\mathit{suspiciousness}_O$ values computed for each statement, the rank is computed as described in Section 2.2.1. Figure 1 shows the OCHIAI suspiciousness and ranks.

3. EXPERIMENT

To understand the impact of various types of test-suite reduction strategies on fault localization, we conducted an experiment. In this section, we first describe the studied variables and measures. We then describe the objects for analysis. Finally, we discuss the details of our experimental setup.

3.1 Variables and Measures

Our primary objective was to investigate the effects of various test-suite reduction strategies on fault-localization effectiveness. Our experiment manipulated two independent variables: the test-suite reduction strategy used to reduce the test suites and the fault-localization technique used to rank the statements in the subject programs. The reduction strategies were described in Section 2.1.2 and the fault-localization techniques were described in Section 2.2. In all, we experimented using 10 test-suite reduction strategies and four fault-localization techniques, resulting in 40 pairings.

For each pairing of a test-suite reduction strategy and a fault-localization technique, we measured two dependent variables: the percentage reduction in the test-suite size and the increase in expense of fault localization. The percentage reduction in test-suite size is measured by calculating the ratio of the size of the reduced test suite to its unreduced test suite. This metric, which we call *Reduction*, is computed by the following equation.

$$\mathit{Reduction} = \left(1 - \frac{\text{size of reduced test suite}}{\text{size of unreduced test suite}}\right) * 100 \quad (7)$$

The effectiveness of the fault-localization technique is measured by the percentage of the program that must be examined to find the fault if using the prescribed rank given by the fault-localization technique. This metric, which we call *Expense*, is computed by the following equation.

$$\mathit{Expense} = \frac{\text{rank of faulty statement}}{\text{number of executable statements}} * 100 \quad (8)$$

An equivalent metric was originally presented by Renieris and Reiss [16] and used by many other researchers (e.g., [6, 11, 14]. For programs that contained multiple faults, we calculated the *Expense* for the first fault to be found as this would relate to the first fault that the developer would begin fixing.

To study the effects of test-suite reduction on fault localization, we compute the *Expense* increase, which is the difference between the *Expense* computed on the reduced test suite and the *Expense* computed on the unreduced test suite, where a positive *Expense* increase means the reduction hurts fault localization while a negative *Expense* increase means the reduction benefits fault localization. Using the *Expense* increase (which is a percentage of the program) allows us to normalize the reduction effects on programs of different sizes.

3.2 Objects for Analysis

We used eight C programs as the objects of analysis (see Table 2). Each program has a variety of versions, each containing one fault. Each program also has a large universe of inputs. We used these C programs because they have been used in many previous fault-localization studies (e.g., [1, 6, 11, 14]). Programs `print_tokens`, `print_tokens2`, `replace`, `schedule`, `schedule2`, `tcas`, and `tot_info`, along with their versions and inputs, were assembled at Siemens Corporate Research for a study of the fault-detection capabilities of control-flow and data-flow coverage criteria [10]. The `space` program was developed by the European Space Agency. A test suite for `space` was constructed from 10,000 test cases generated randomly by Vokolos and Frankl [17] and 3,585 test cases created by researchers in the Aristotle Research Group [3]. Each version of the Siemens programs and each original version of the `space` program contains exactly one fault, although the faults may span multiple statements or even functions. In addition to the single-fault versions, we randomly generated 10 2-fault versions and 10 3-fault versions for the `space` program by injecting the faults from its original versions into the version that is deemed to have no faults—the correct version.

Table 2: Objects of Analysis.

Program	Faulty Versions	LOC	Test Cases	Description
print_tokens	7	472	4056	lexical analyzer
print_tokens2	10	399	4071	lexical analyzer
replace	32	512	5542	pattern replacement
schedule	9	292	2650	priority scheduler
schedule2	10	301	2680	priority scheduler
space	58	6218	13585	array definition interpreter
tcas	41	141	1578	altitude separation
tot_info	23	440	1054	information measure

Combined, there are 190 faulty versions. Of these versions, we were able to use 169 versions. Two versions—versions 4 and 6 of `print_tokens`—contained no syntactic differences from the correct version of the program in the C file—there were only differences in a header file. In eight versions—version 32 of `replace`, version 9 of `schedule2`, and versions 1, 2, 3, 12, 32 and 34 of `space`—no test cases fail, thus the fault was never manifested. In 11 versions—version 10 of `print_token2`, version 27 of `replace`, versions 5, 6, and 9 of `schedule`, and versions 25, 26, 30, 35, 36, and 38 of `space`—test cases failed because of a segmentation fault. Thus, we were unable to use these 11 versions for our experiment. After removing the 21 versions, we were left with the 169 versions.

3.3 Experimental Setup

We applied the 10 test-suite reduction strategies and the four fault-localization techniques to the 169 versions of our programs and their test suites. This section describes the way in which we set up the experiment to apply the test-suite reduction strategies and the fault-localization techniques that we used.

We used three steps to set up the experiment. First, to simulate realistically-sized test suites for these programs and to experiment with test suites of different composition, for each of the 169 versions, we randomly generated 10 test suites of different sizes containing from 50 test cases to 500 test cases by increasing the test suite size by 50 test cases each time. This process created $1,690$ ($169 * 10$) test suites with sizes ranging from 50 to 500. To provide an average over many test suites, we repeated the first step 100 times, which created $169,000$ ($1,690 * 100$) test suites. We used these $169,000$ test suites as the unreduced test suites. Second, we applied the 10 reduction strategies from Section 2.1.2 to the unreduced test suites. This gave us $1,690,000$ ($169,000 * 10$) reduced test suites. Including the $169,000$ unreduced test suites with the $1,690,000$ reduced test suites resulted in $1,859,000$ test suites of different sizes. Third, we applied the four fault-localization techniques to the $1,859,000$ test suites and recorded the $7,436,000$ ($1,859,000 * 4$) fault-localization results for the analysis.

3.3.1 Generating Unreduced Test Suites

Each version of the subject programs that we used has a large test pool. We used its entire test pool as the input and applied the following process to randomly generate the unreduced test suites.

1. We randomly selected one failed test case from the test pool to ensure that the generated test suite has at least one failed test case.
2. We randomly selected one test case from the test pool (without considering its pass/fail status) and repeated this process until we got the desired number (e.g., $50, 100, \ldots$) of test cases in the test suites. Each time one test case was selected, we marked it so that it was not selected again.

3.3.2 Applying Reduction Strategies

For each of the $169,000$ unreduced test suites, we used the following process to apply the five statement-based reduction strategies.

1. We marked all statements as "uncovered" and all test cases as "unselected."
2. For each "unselected" test case, we calculated the number of "uncovered" statements that it covered.
3. We marked the first (if there was more than one) test case encountered that covered the maximum number of statements as "selected," and we marked all statements it covered as "covered."[5]
4. We repeated Steps 1-3 until there were no remaining "uncovered" statements covered by any "unselected" test cases.
5. We considered all "selected" test cases as members of the reduced test suite.

Similarly, for each of the $169,000$ unreduced test suite, we used the following process to apply the five vector-based reduction strategies.

1. We iterated over the test cases in the test suite, checking, for each test case, whether we have already encountered this exact set of statements (or vector) covered by another test case. If we have not encountered it before, we created a bin for it and placed that test case in that bin. If we have encountered it before, we placed the test case in the matching bin.
2. We randomly selected one test case from each bin. The test cases that were selected comprised the reduced test suite.

Specially, for the strategies SA and VA, we randomly selected one failed test case first to ensure that the reduced test suite has at least one failed test case. Otherwise, fault-localization may not be needed or applied.

4. DATA AND ANALYSIS

In this section, we describe the results of our experiment, summarize and explain the data, and discuss threats to the validity of the experiment and how we addressed them.

[5]This is equivalent to randomly selecting one test case because the test suites we used were randomly generated from Section 3.3.1.

4.1 Experimental Results

We organize the presentation of the experimental results in the following way. We first examine the effects of all 10 test-suite reduction strategies on one of the fault-localization techniques—TARANTULA. Section 4.1.1 presents the results of the way in which applying each of the 10 test-suite reduction strategies affects TARANTULA's fault-localization effectiveness. We next present the reduction achieved by each of the 10 test-suite reduction techniques. These results are important because they show that the sizes of the reduced and unreduced test suites differ. Section 4.1.2 presents these test-suite reduction results. Based on the results of Sections 4.1.1 and 4.1.2, we chose two reduction strategies that are representative of the others, and present their effects on each of the four fault-localization techniques. Section 4.1.3 presents these results. Finally, in Section 4.1.4, we show the size reduction of the test suites by the two representative reduction strategies for each of the subject programs.

4.1.1 Expense Increase on TARANTULA

Table 3 shows the increase in Expense of using the TARANTULA fault-localization technique on all 10 test-suite reduction strategies for the eight single-fault programs. In the table, rows represent the subject programs and columns represent the test-suite reduction strategies using their abbreviations. Each entry in the table represents the mean of the *Expense* (see Equation 8) increase over the base *Expense* computed on the unreduced test suite. The mean is computed over all versions of the program, over all 100 iterations, and over all 10 differently-sized test suites. For example, for `replace`, the mean increase in expense over the unreduced test suite for test-suite reduction strategy SP is 3.958. The last row in the table is a summary aggregation over all versions, and is computed as the mean over all versions of all of the programs, over all 100 iterations, and over all 10 differently-sized test suites.

The table shows that all statement-based reduction strategies incur a greater expense increase than the vector-based strategies. Although there are a few exceptions using the SP strategy, the overwhelming trend is that these statement-based reduction strategies cause an increase in the expense. This means that, for our subject programs, if a test suite is reduced using statement-based strategies, the fault-localization technique will almost always perform worse. Among the statement-based reduction strategies, for the subjects we studied, reducing on all test cases with preference for failed (SR) causes the greatest increase in fault-localization expense, and reducing on all failed test cases (SF) causes the least increase in expense. Among the vector-based reduction strategies, reducing on any of the unreduced test suites shows a negligible impact on the fault-localization expense. This means that, for our subject programs, if a test suite is reduced using vector-based strategies, the fault-localization technique will almost always perform the same. We also see that for vector-based strategies, reducing on the failed test cases (VF) incurs the greatest increase on average and reducing on all test cases with preference for failed (VR) causes the least increase in the fault-localization expense. In fact, on many versions and programs, and overall, reducing based on the VR strategy causes a decrease in the fault-localization techniques' expense, although we note that this decrease is small and not always present.

Table 3: Mean Increase in Fault-localization Expense using Tarantula on Reduced Test Suites.

	SA	SP	SF	SPF	SR	VA	VP	VF	VPF	VR
print_tokens	4.934	1.707	1.418	3.257	9.824	0.062	-0.024	0.077	0.038	0.031
print_tokens2	4.597	-0.397	1.047	1.288	12.674	-0.408	-0.435	0.005	-0.433	-0.452
replace	4.747	3.958	0.330	4.339	12.344	0.310	0.287	0.011	0.298	0.296
schedule	8.805	7.573	0.256	9.563	16.367	-0.367	-0.044	-0.387	-0.369	-0.356
schedule2	6.081	5.423	1.282	6.738	7.429	0.600	0.791	-0.071	0.728	0.739
space	0.024	-0.243	0.313	-0.038	1.265	-0.005	-0.014	0.014	0.000	-0.010
tcas	6.854	7.127	0.225	7.047	9.014	0.019	-0.037	0.182	0.071	0.072
tot_info	4.895	1.656	1.252	3.117	6.043	-1.075	-0.828	-0.056	-1.203	-1.222
Summary	4.767	3.637	0.575	4.266	8.322	-0.100	-0.063	0.029	-0.102	-0.107

Table 4: Mean Percentage of Test-Suite Size Reduction using the 10 Reduction Strategies.

	SA	SP	SF	SPF	SR	VA	VP	VF	VPF	VR
print_tokens	96.654	94.904	0.813	95.717	96.804	24.330	24.265	0.045	24.310	24.330
print_tokens2	97.123	91.018	4.540	95.558	97.369	28.006	27.562	0.300	27.861	28.006
replace	94.426	92.310	1.177	93.488	94.666	25.939	25.594	0.262	25.856	25.939
schedule	97.657	92.848	4.388	97.235	97.863	54.678	50.353	2.920	53.272	54.678
schedule2	97.495	96.046	0.925	96.970	97.796	35.702	35.363	0.166	35.529	35.702
space	70.286	55.622	12.443	68.066	70.388	12.010	3.843	8.148	11.991	12.010
tcas	97.728	94.840	2.146	96.986	97.749	95.100	92.402	2.114	94.516	95.100
tot_info	97.043	88.273	7.023	95.296	97.063	68.241	64.054	3.438	67.493	68.241
Summary	92.079	86.238	4.617	90.855	92.201	50.887	47.780	2.734	50.514	50.887

4.1.2 Percentage Reduction

Table 4 shows the percentage reduction in the size of the test suite using each of the 10 test-suite reduction strategies. Like Table 3, rows represent the subject programs and columns represent the test-suite reduction strategies using their abbreviations. Each entry in the table is the mean of the percentage reduction of the test suite from the unreduced test suite using the indicated strategy. For example, for `replace`, the mean reduction of 92.310% is achieved on the unreduced test suite by test-suite reduction strategy *SP*; this means that the reduced test suite is only 7.690% of the unreduced test suite. Each mean is computed over all faulty versions of the program, over all 100 iterations, and over all 10 differently-sized test suites. The last row in the table is a summary, and is computed as the mean over all versions of all programs, over all 100 iterations, and over all 10 differently-sized test suites.

From the table, we can see that most statement-based reduction strategies provide more reduction than the vector-based strategies. On average, the statement-based reduction strategies provide about a 90% reduction in the test-suite size, and the vector-based reduction strategies provided about a 50% reduction in the test-suite size. One exception occurs for both statement-based and vector-based reduction when they are applied to the failed test base: the statement-based reduction strategy applied to only failed test cases (*SF*) provides only about 5% reduction, and the vector-based reduction strategy applied to only failed test cases (*VF*) provides only about 3% reduction. This small reduction occurs because, in general, these test suites contain many more passed test cases than failed test cases, and thus, less reduction is achieved when reducing only on these relatively few failed test cases. Additionally, vector-based reduction strategies provides only about 10% reduction on `space` because, on average, about 60% of the 13585 test cases generate unique statement vectors. When we randomly sampled these test cases to create the relatively small unreduced test suites ($\leq$ 500 test cases), a high percentage of the test cases in these suites had unique vectors.

4.1.3 Expense Increase on All Fault-localization Techniques

To evaluate and compare the effects of test-suite reduction on all four fault-localization techniques discussed in Section 2, we present the results of each fault-localization technique on two strategies: statement-based reduction on all test cases (*SA*) and vector-based reduction on all test cases (*VA*).

Figure 2 shows these results. We present the data using 10 boxplot[6] charts. Figure 2 shows the charts for each of the subject programs. Each boxplot column shows a fault-localization technique applied to a reduced test suite produced by either statement-based reduction or vector-based reduction. The fault-localization techniques are abbreviated as such: *T* for Tarantula, *S* for Statistical Bug Isolation, *J* for Jaccard, and *O* for Ochiai.

This data shows that, for our subject programs, these test-suite reduction strategies have a similar effect on all four fault-localization techniques for each subject program. The data also shows that the statement-based reduction strategy clearly produces both a greater increase in fault-localization expense and greater variability in those increases over the vector-based strategy. Whereas the boxplots for *SA* are generally raised and wide, the boxplots for *VA* are centered at zero and narrow.

4.1.4 Size Results

Figure 3 shows the percentage reduction for each test-suite reduction strategy on each subject program as two boxplot charts. The left chart shows the results for the *SA* strategy and the right chart shows the results for the *VA* strategy. The vertical axis for these charts represents the percentage of test-suite size reduction for each program and reduction strategy. The figure shows that the statement-based reduc-

[6]A boxplot is a standard statistical device for representing data sets. In these boxplots, each data set's distribution is represented by a box. The box's height spans the central 50% of the data and its upper and lower ends mark the upper and lower quartiles. The middle of the three horizontal lines within the box represents the median. The vertical lines attached to the box indicate the tails of the distribution.

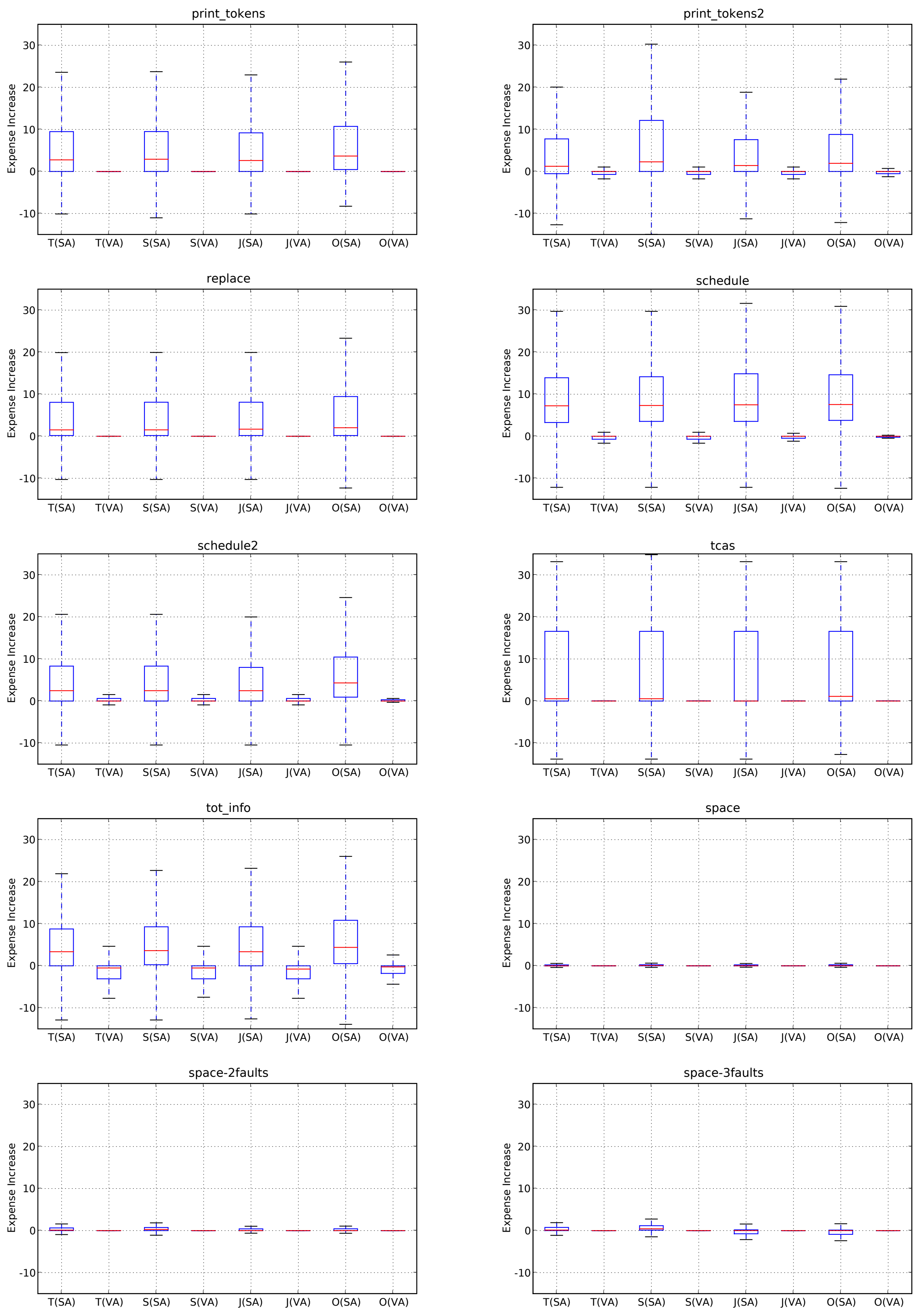

Figure 2: Expense increase for the statement-based reduction (SA) and the vector-based reduction (VA).

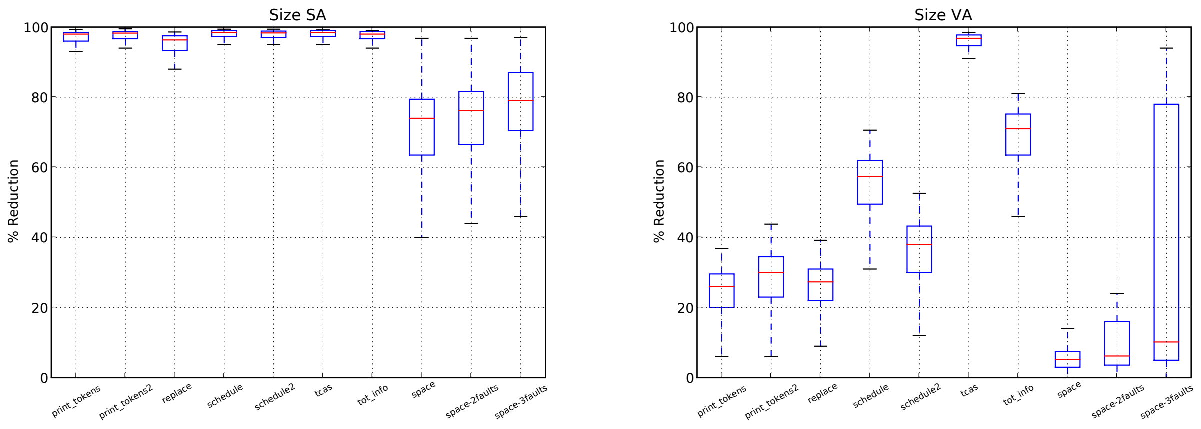

Figure 3: Percentage of test-suite size reduction for statement-based reduction (SA) and vector-based reduction (VA).

tion strategy provides a much greater and more consistent reduction than the vector-based reduction strategy.

4.2 Discussion

In this section, we summarize and provide some observations about the results that we obtained.

The data demonstrate a trade-off between the test-suite reduction that is achieved and the effectiveness of the fault localization. The statement-based reduction strategy provides much greater reduction of the test suites but in general negatively affects the effectiveness of the fault-localization techniques. The vector-based reduction provided less reduction in test-suite size, but provides negligible impact on the effectiveness of the fault-localization techniques. These results hold for all four fault-localization techniques.

In their study, Hao and colleagues [7] found that test-case redundancy can negatively affect fault localization. Our studies provide a more thorough experiment with the goal of of investigating whether removing redundancy from the test suite improves the effectiveness of fault-localization techniques, as they proposed. Our evaluation does not support their finding that redundancy is a major source of fault-localization error. Although we observed that occasionally the fault localization improves by removing redundancy, the improvement is small and unpredictable.

Given that our experiment shows that, for our subjects, elimination of test-suite redundancy generally negatively impacts effectiveness of fault localization, we were interested in whether it is possible to retain fault-localization effectiveness, with negligible impact, while saving testing costs. We observed that usually traditional, statement-based reduction can save testing expense, but it comes at the cost of effectiveness of fault localization. We investigated a stricter reduction criterion—vector-based—and showed that in general, for our subject programs, testing expense could be reduced with negligible effects on fault-localization effectiveness.

Because of the trade-off between reduction and fault-localization effectiveness, we recommend that developers utilize the reduction strategy according to the time that can be allocated to testing. If testing time is limited, testing cost is very high, or developer time is inexpensive, the statement-based reduction strategy may be most appropriate. If developer time is most important, the vector-based reduction strategy may be most appropriate. Additionally, if testing cost is inexpensive, then the entire test suite may be run to provide the fault-localization technique with the most information.

4.3 Threats to Validity

Threats to internal validity arise when factors affect the dependent variables without the researchers' knowledge. It is possible that some implementation flaws could have affected the results. However, we are confident in the accuracy of the results, given that we implemented four fault-localization techniques and 10 test-suite reduction strategies, and the results were consistent among them.

Threats to external validity arise when the results of the experiment are unable to be generalized to other situations. In this experiment, we evaluated the effects of test suite reduction on fault localization using only eight programs, and thus, we are unable to definitively state that our findings will hold for programs in general. We attempted to address some of these uncertainties by performing our evaluation on a variety of programs of varying size. For each subject program, we performed our evaluation on varying sizes of test suites, many different faults, and many randomly chosen test suites. We also performed evaluation on a varying number of faults for one of the programs to demonstrate how this factor affects the results. In addition, we implemented and evaluated the effects on four fault localization techniques.

Threats to construct validity arise when the metrics used for evaluation do not accurately capture the concepts that they are meant to evaluate. In our case, we measure the effectiveness of the fault localization techniques using the *Expense* measure that shows the percent of the code that must be examined to find the fault. The metric assumes that the developer will inspect the program, statement by statement, in the prescribed order until reaching the fault, and that she will be able to recognize that it is faulty. While this may not be a realistic debugging process, we believe that it is a reasonable approximation of relative effectiveness of the fault localization technique. For example, a technique that identifies the fault as the most suspicious statement will likely provide the developer with a better hint than another technique that marks the fault as the least suspicious statement.

5. SUMMARY AND CONCLUSIONS

Testing and debugging costs often dominate the software development process, and thus, researchers have proposed test-suite reduction techniques and fault-localization techniques to address each of these sources of expense. However, to date, there has been little investigation of how these two cost-saving techniques affect each other. In this paper, we provide the first investigation of the effects of test-suite reduction on fault-localization effectiveness. We evaluated several test-suite reduction strategies on multiple fault-localization techniques, and proposed new reduction strategies that help to leverage the trade-offs between reduction and localization effectiveness. We found that traditional statement-based reduction negatively affects the fault localization, whereas vector-based reduction provides negligible affects on the effects of fault localization. Moreover, occasionally the vector-based reduction allows for an improvement for the fault localization techniques.

Although our evaluation clearly demonstrates a trade-off between reduction and fault-localization effectiveness for our subject programs, more experimentation must be conducted to verify the effects in general. Specifically, we are planning experiments on larger programs. We also plan to experiment with other reduction strategies that reduce with respect to other coverage criteria such as branches, dataflows, and paths. We also would like to explore strategies that are more strict than the statement-based reduction but less strict than the vector-based reduction to provide more alternatives in the trade-off between reduction and fault-localization effectiveness.

6. ACKNOWLEDGEMENTS

This work was supported in part by National Science Foundation awards under CCR-0205422, CCF-0429117, and CCF-0541049 to Georgia Tech. The anonymous reviewers provided many helpful suggestions to improve the paper.

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] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In *Proceedings of IEEE Software Reliability Engineering*, pages 143–151, 1995.

[3] Aristotle Research Group. ARISTOTLE analysis system, 2007. `http://www.cc.gatech.edu/aristotle/`.

[4] B. Baudrey, F. Fleurey, and Y. L. Traon. Improving test suites for efficient fault localization. In *International Conference on Software Engineering*, pages 82–91, Shanghai, China, May 2006.

[5] T. Y. Chen and M. F. Lau. Dividing strategies for the optimization of a test suite. *Information Processing Letters*, 60(3):135–141, Mar. 1996.

[6] 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.

[7] D. Hao, Y. Pan, L. Zhang, W. Zhao, H. Mei, and J. Sun. A similarity-aware approach to testing based fault localization. In *Proceedings of the Conference on Automated Software Engineering*, pages 291–294, November 2005.

[8] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

[9] J. R. Horgan and S. A. London. ATAC: A data flow coverage testing tool for C. In *Proceedings of the Symp. on Assessment of Quality Software Development Tools*, pages 2–10, May 1992.

[10] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the International Conference on Software Engineering*, pages 191–200, May 1994.

[11] 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.

[12] 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.

[13] 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*, pages 15–26, 2005.

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

[15] J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage-based test sets. In *Proceedings of the International Conference on Testing Comp. Software*, pages 111–123, June 1995.

[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] F. Vokolos and P. Frankl. Empirical evaluation of the textual differencing regression testing techniques. In *Proceedings of the International Conference on Software Maintenance*, November 1998.