Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique

James A. Jones and Mary Jean Harrold
College of Computing, Georgia Institute of Technology
Atlanta, Georgia, U.S.A.
jjones@cc.gatech.edu, harrold@cc.gatech.edu

Abstract

The high cost of locating faults in programs has motivated the development of techniques that assist in fault localization by automating part of the process of searching for faults. Empirical studies that compare these techniques have reported the relative effectiveness of four existing techniques on a set of subjects. These studies compare the rankings that the techniques compute for statements in the subject programs and the effectiveness of these rankings in locating the faults. However, it is unknown how these four techniques compare with Tarantula, another existing fault-localization technique, although this technique also provides a way to rank statements in terms of their suspiciousness. Thus, we performed a study to compare the Tarantula technique with the four techniques previously compared. This paper presents our study—it overviews the Tarantula technique along with the four other techniques studied, describes our experiment, and reports and discusses the results. Our studies show that, on the same set of subjects, the Tarantula technique consistently outperforms the other four techniques in terms of effectiveness in fault localization, and is comparable in efficiency to the least expensive of the other four techniques.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Experimentation

Keywords

Fault localization, automated debugging, program analysis, empirical study

1. INTRODUCTION

Debugging software is an expensive and mostly manual process. Of all debugging activities, locating the faults, or *fault localization*, is the most expensive [14]. This expense occurs in both the time and the cost required to find the fault. Because of this high cost, any improvement in the process of finding faults can greatly decrease the cost of debugging.

In practice, software developers locate faults in their programs using a highly involved, manual process. This process usually begins when the developers run the program with a test case (or test suite) and observe failures in the program. The developers then choose a particular failed test case to run, and iteratively place breakpoints using a symbolic debugger, observe the state until an erroneous state is reached, and backtrack until the fault is found. This process can be quite time-consuming.

To reduce the time required to locate faults, and thus the expense of debugging, researchers have investigated ways of helping to automate this process of searching for faults. Some existing techniques use coverage information provided by test suites to compute likely faulty statements (e.g., [1, 4, 5, 10, 12]). Other techniques perform a binary search of the memory state using one failing test case and one passing test case to find likely faulty statements (e.g., [2, 15]). Still other techniques are based on the remote monitoring and statistical sampling of programs after they are deployed (e.g., [6, 7, 8, 9]). Most papers reporting these techniques also report empirical studies that evaluate the presented technique in terms of its effectiveness and efficiency. However, because of the difficulty in comparing techniques developed on different platforms for different languages and using different programs and test suites, few empirical studies have reported comparison of existing techniques.

Although comparing the fault localization of techniques is difficult, several recent studies have compared four existing techniques in terms of their ability to localize faults. Renieris and Reiss [12] presented their technique, called *Nearest Neighbor*, and compared it to two techniques that use set union and intersection operations on coverage data (similar to those presented in [1] and [11]). Their studies show that, on a given set of subjects, the Nearest-Neighbor technique performs more effectively than the two set-based approaches. Later, Cleve and Zeller [2] presented their technique, called *Cause Transitions*, compared it to the Nearest-Neighbor technique, and found that, on the same set of subjects, Cause Transitions consistently performs better than Nearest Neighbor (and thus set union and intersection tech-

ASE'05, November 7–11, 2005, Long Beach, California, USA.
Copyright 2005 ACM 1-58113-993-4/05/0011 ...$5.00.

niques) in terms of effectiveness. Because of these results, they conclude that Cause Transitions locates faults twice as well as other fault-localization techniques [2, p. 350]. However, their studies do not include a comparison of the *Tarantula* technique—another existing technique [5] that also computes a *ranking* of statements in terms of their suspiciousness. Thus, it is unknown how these four techniques compare with Tarantula.

Our preliminary evaluation of the Tarantula technique suggested that it would perform better than both the Nearest-Neighbor and the Cause-Transitions techniques, and thus the others considered in these previous studies. To confirm this, we performed an empirical study to compare the fault-localization ability of Tarantula with the fault-localization ability of these other techniques on the same set of subjects.

In this paper, we present this empirical study. We first present an overview of the techniques that our study compares. We present the Tarantula approach in some detail, and explain how its results, originally presented in visual form, can also be reported as a ranking of statements in terms of their likelihood of faultiness; we do this to provide a fair comparison with the other techniques. We present a brief overview of the other four techniques.

We then present the details of the study, which compares the techniques in terms of efficiency and effectiveness. The results of our study show that, for the set of subjects studied, Tarantula consistently outperforms these other four techniques in terms of effectiveness and is comparable in terms of cost with the most efficient techniques.

The main contributions of the paper are:

- The first study that compares the fault-localization ability of Tarantula with Set union, Set intersection, Nearest Neighbor, and Cause Transitions. Our study shows that, for the subjects studied, Tarantula consistently outperforms these techniques, making it the best existing technique known for fault localization on these subjects.

- A description of the Tarantula approach in terms of ranking of statements for their suspiciousness that provides a way to compare it with the Nearest-Neighbor and Cause-Transitions techniques, as well as other future techniques.

- A compilation of existing fault-localization results that have been conducted on the same set of subjects. This compilation provides a survey of existing empirical studies on these subjects, and can be used as a benchmark for future fault-localization studies.

2. FAULT-LOCALIZATION TECHNIQUES STUDIED

In this section, we present an overview of the five fault-localization techniques we studied—Tarantula, Set union, Set intersection, Nearest Neighbor, and Cause Transitions. For each technique, we describe (1) its method for computing an initial set of suspicious statements in the program—the set of statements where the search for the fault should begin and (2) its method of ordering (or ranking) the rest of the statements for continuing the search in case the fault is not found in this initial set of suspicious statements.

2.1 Tarantula

Software testers often gather large amounts of data about a software system under test. These data can be used to demonstrate the exhaustiveness of the testing, and find areas of the source code not executed by the test suite, thus prompting the need for additional test cases. These data can also provide information that can be useful for fault localization.

Tarantula utilizes such information that is readily available from standard testing tools: the pass/fail information about each test case, the entities that were executed by each test case (e.g., statements, branches, methods), and the source code for the program under test. The intuition behind Tarantula is that entities in a program that are primarily executed by failed test cases are more likely to be faulty than those that are primarily executed by passed test cases. Unlike most previous techniques that used coverage information (e.g., [1]), Tarantula allows some tolerance for the fault to be occasionally executed by passed test cases. We have found that this tolerance often provides for more effective fault localization.

Previously, we presented our Tarantula technique [4, 5] and a visualization tool that uses the technique for assigning a value for each program entity's likelihood of being faulty. We did this by specifying a color for each statement in the program. We utilize a color (or hue) spectrum from red to yellow to green to color each statement in the program under test. The intuition is that statements that are executed primarily by failed test cases and are thus, highly suspicious of being faulty, are colored red to denote "danger"; statements that are executed primarily by passed test cases and are thus, not likely to be faulty, are colored green to denote "safety"; and statements that are executed by a mixture of passed and failed test cases and thus, and do not lend themselves to suspicion or safety, are colored yellow to denote "caution."

In particular, the hue of a statement, s, is computed by the following equation:

$$hue(s) = \frac{\frac{passed(s)}{totalpassed}}{\frac{passed(s)}{totalpassed} + \frac{failed(s)}{totalfailed}} \quad (1)$$

In Equation 1, $passed(s)$ is the number of passed test cases that executed statement s one or more times. Similarly, $failed(s)$ is the number of failed test cases that executed statement s one or more times. $totalpassed$ and $totalfailed$ are the total numbers of test cases that pass and fail, respectively, in the entire test suite. Note that if any of the denominators evaluate to zero, we assign zero to that fraction. Our Tarantula tool used the color model based on a spectrum from red to yellow to green. However, the resulting $hue(s)$ can be scaled and shifted for other color models.

Although we expressed these concepts in the form of statement coloring, they compute values that can be used without visualization. The $hue(s)$ is used to express the likelihood that s is faulty, or the *suspiciousness* of s. The $hue(s)$ varies from 0 to 1 — 0 is the most suspicious and 1 is the least suspicious. To express this in a more intuitive manner where the value increases with the suspiciousness, we can either subtract it from 1, or can equivalently replace the numerator with the ratio of the failed test cases for s. Also, note that we can define this metric for other coverage entities such as branches, functions, or classes.

	Test Cases							

mid() {
int x,y,z,m;
	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	suspiciousness	rank

1: read("Enter 3 numbers:",x,y,z);	●	●	●	●	●	●	0.5	7
2: m = z;	●	●	●	●	●	●	0.5	7
3: if (y<z)	●	●	●	●	●	●	0.5	7
4: if (x<y)	●	●			●	●	0.63	3
5: m = y;		●					0.0	13
6: else if (x<z)	●				●	●	0.71	2
7: m = y; // *** bug ***	●					●	0.83	1
8: else			●	●			0.0	13
9: if (x>y)			●	●			0.0	13
10: m = y;			●				0.0	13
11: else if (x>z)				●			0.0	13
12: m = x;							0.0	13
13: print("Middle number is:",m);	●	●	●	●	●	●	0.5	7
} Pass/Fail Status	P	P	P	P	P	F		

Figure 1: Example of Tarantula technique.

With these simple modifications, we define the suspiciousness of a coverage entity e with the following equation:

$$\begin{aligned} suspiciousness(e) &= 1 - hue(e) = \\ &= \frac{\frac{failed(e)}{totalfailed}}{\frac{passed(e)}{totalpassed} + \frac{failed(e)}{totalfailed}} \end{aligned} \qquad (2)$$

Using the *suspiciousness* score, we sort the coverage entities of the program under test. The set of entities that have the highest suspiciousness value is the set of entities to be considered first by the programmer when looking for the fault. If, after examining these statements, the fault is not found, the remaining statements should be examined in the sorted order of the decreasing suspiciousness values. This specifies a *ranking* of entities in the program. For evaluation purposes, each set of entities at the same ranking level is given a rank number equal to the greatest number of statements that would need to be examined if the fault were the last statement in that rank to be examined. For example, if the initial set of entities is ten statements, then every statement in that set is considered to have a rank of 10.

To illustrate how the Tarantula technique works, we provide a simple example program, *mid()*, and test suite, given in Figure 1. Program *mid()* takes three integers as input and outputs the median value. The program contains a fault on line 7—this line should read "`m = x;`". To the right of each line of code is a set of six test cases: their input is shown at the top of each column, their coverage is shown by the black dots, and their pass/fail status is shown at the bottom of the columns. To the right of the test case columns are two columns labeled "suspiciousness" and "rank." The suspiciousness column shows the suspiciousness score that the technique computes for each statement. The ranking column shows the maximum number of statements that would have to be examined if that statement were the last statement of that particular suspiciousness level chosen for examination. The ranking is ordered on the suspiciousness, from the greatest score to the least score.

Consider statement 1, which is executed by all six test cases containing both passing and failing test cases. The Tarantula technique assigns statement 1 a suspiciousness score of 0.5 because one failed test case executes it out of a total of one failing test case in the test suite (giving a ratio of 1), and five passed test cases execute it out of a total of five passing test cases in the test suite (giving a ratio of 1). Using the suspiciousness equation specified in Equation 2, we get $1/(1+1)$, or 0.5. When Tarantula orders the statements according to suspiciousness, statement 7 is the only statement in the initial set of statements for the programmer to inspect. If the fault were not at line 7, she would continue her search by looking at the statements at the next ranks. There are three statements that have higher suspiciousness values than statement 1. However, because there are four statements that have a suspiciousness value of 0.5, Tarantula assigns every statement with that suspiciousness value a rank of 7 (3 statements examined before, and a maximum of 4 more to get to statement 1). Note that the faulty statement 7 is ranked first—this means that programmer would find the fault at the first statement that she examined.

2.2 Set Union and Set Intersection

Several researchers have used coverage based information for fault localization. Agrawal and colleagues present a technique that computes the set difference of the statements covered by two test cases—one passing and one failing [1]. A set of statements is obtained by removing the statements executed by the passed test case from the set of statements executed by the failed test case. This resulting set of statements is then used as the initial set of suspicious statements when searching for faults.

Pan and colleagues present a set of dynamic-slice-based heuristics that use set algebra of test cases' dynamic slices for similar purposes [11]. In addition to using the information listed above, they also use analysis information to compute dynamic slices of the program and test cases from a particular program point and particular variable.

Some simple and common techniques described in [12] for computing a subset of all of coverage entities[1] are the *Set-union* and *Set-intersection* techniques. The Set-union technique computes a set by removing the union of all statements executed by all passed test cases from the set of statements executed by a single failed test case. That is, given a set of passing test cases P containing individual passed test cases p_i, and a single failing test case f, the set of coverage entities executed by each p is E_p, and the coverage entities executed by f is E_f. The union model gives

$$E_{initial} = E_f - \bigcup_{p \in P} E_p \qquad (3)$$

The Set-intersection technique computes the set difference between the set of statements that are executed by every passed test case and the set of statements that are executed by a single failing test case. A set of statements is obtained by intersecting the set of statements executed by all passed test cases and removing the set of statements executed by the failed test case. The intuition is to give the statements that were neglected to be run in the failed test case, but were run in every passed test case. Using the same notation as Equation 3, we can express the Set-intersection technique as

$$E_{initial} = \bigcap_{p \in P} E_p - E_f \qquad (4)$$

The resulting set $E_{initial}$ for each of these two techniques defines the entities that are suspected of being faulty. In searching for the faults, the programmer would first inspect these entities. To illustrate the Set-union and Set-intersection techniques, consider their application to program *mid()* and test suite given in Figure 1. For this example, both techniques compute an empty initial set of statements. Thus, for this example, these techniques would fail to assist in fault localization. To demonstrate how these techniques could work on a different example, consider the same program, but with the test suite consisting of test cases 2-6 (i.e., omitting the first test case in the test suite). When we apply the Set-union method, the set of statements in the union of all passed test cases consists of statements 1, 2, 3, 4, 5, 6, 8, 9, 10, 11 , and 13. When we remove these statements from the the set of statements executed by the failed test case, we are left with an initial set containing only one program entity—statement 7. In this case, the Set-union technique would have identified the fault in the initial set. However, notice the sensitivity of this technique to the particular test cases used—for many test suites, the initial set is either the null set or fails to include the fault.

If the fault is not found in the initial set of entities computed by the set-based approaches, there must be a strategy to guide the programmer's inspection of the rest of the statements in the program. Renieris and Reiss suggest a technique that provides an ordering to the entities based on the system dependence graph, or SDG [12]. We refer to this ranking technique as the *SDG-ranking technique*. Under this ranking technique, nodes that correspond to the initial set of entities are identified; they call these *blamed* nodes. A breadth-first search is conducted from the blamed nodes along dependency edges in both forward and backward directions. All nodes that are at the same distance are grouped together into a single rank. Every node in a particular rank is assigned a rank number, and this number is the same for all constituent nodes in the rank. Given a distance d, and a set of nodes at that distance $S(d)$, the rank number that is assigned to every node in $S(d)$ is the size of every set of nodes at lesser distances plus the size of $S(d)$.

For example, consider a scenario where an initial set contains three statements. These three statements correspond to three nodes in the SDG. The programmer inspects these statements and determines that the fault is not contained in them. She then inspects all forward and backward controlflow and dataflow dependencies at a distance of 1. This gives an additional seven nodes. The rank number of all nodes in the initial set is 3, and the rank number of all nodes at a distance of 1 is 10 (i.e., $(3+7)$). Using the size of the rank plus the size of every rank at a lesser distance for the rank number gives the maximum number of nodes that would have to be examined to find the fault following the order specified by the technique. This is similar to the use of the maximum rank number described in the Tarantula technique.

We [5] and others [12] have found that these set-based coverage techniques often perform poorly. One reason for this is that most faulty statements are executed by some combination of both passed and failed test cases. However, when using set operations on coverage-based sets, the faulty statement is often removed from the resulting set of statements to be considered; the application of the Set-union technique to our example illustrates this. These techniques' ineffectiveness when faults are executed by occasional passed test cases was recognized by us and other researchers, and this recognition motivated techniques that allow some tolerance for these cases.

2.3 Nearest Neighbor

Renieris and Reiss [12] address the issue of tolerance for an occasional passed test case executing a fault with their Nearest-Neighbor technique. Rather than removing the statements executed by all passed test cases from the set of statements executed by a single failed test case, they selectively choose a single best passed test case for the set difference. By removing the set of statements executed by a passed test case from the set of statements executed by a failed test case, their approach applies the technique of Agrawal and colleagues in [1], but has a specific technique for specifying which passed test case to use for this set difference. They choose any single failed test case and then find the passed test case that has coverage that is most similar to the coverage of the failed test case. Utilizing these two test cases, they remove the set of statements executed by the passed test case from the set of statements executed by the failed test case. The resulting set of statements is the initial set of statements from which the programmer should start her search for the fault.

Renieris and Reiss defined two measures for the similarity of the coverage sets between the passed and failed test cases.

[1] Coverage entities are program entities, such as statements, branches, functions, and classes, that can be instrumented and covered by a test case.

They call the first measure *binary distancing.* This measure computes the set difference of the set of statements covered by the chosen failed test case and the set of statements covered by a particular passed test case. They propose that this measure could be defined as either (1) the cardinality of the symmetric set difference of the statements executed by each of the passing and failing test cases, or (2) the cardinality of the asymmetric set difference between the set of statements executed by the failed test case and the set of statements executed by the passed test case. They call their second measure *permutation distancing.* In this measure, for each test case, a count is associated with each statement or basic block that records the number of times it was executed by the test case. The statements are then sorted by the counts of their execution. The permutation distance measure of two test cases is based on the cost of transforming one permutation to the other.

After an arbitrary failed test case is chosen, the distance value is computed for every passed test case. The passed test case that has the least distance is chosen. They then remove the set of statements executed by this passed test case from the set of statement executed by the failed test case. This resulting set is the initial set of statements for the programmer to examine to find the fault.

If the fault is not contained in the initial set, they specify using the SDG-ranking technique (presented in Section 2.2) on the remaining nodes starting at the initial set. The remaining program points should be examined in the order specified by the ranking technique.

To illustrate how this technique works, consider our example program, *mid()* and its test suite presented in Figure 1. In this test suite, only one failed test case exists, thus we choose it as our base for measuring distances. The distance is measured for every test case in the suite and the first test case is chosen as the test case with the least distance—it covers exactly the same set of statements as the failed test case. When we remove the set of statements executed by the passed test case from the set of statements executed by the failed test case, we get the null set as our initial set of statements to examine. Thus, for this test suite and program, this technique is ineffective. To demonstrate how this technique could work on a different example, consider the same program, but with the test suite consisting of test cases 2-6 (i.e., omitting the first test case in the test suite). We find that the fifth test case is the passed test case with the least distance. When we remove the set of statements executed by the fifth test case from the set of statements executed by the failed test case, we obtain a set containing only statement 7. In this case, the Nearest-Neighbor technique would have identified the fault in the initial set. However, notice that this technique is also sensitive to the particular test cases used.

2.4 Cause Transitions

Cleve and Zeller's Cause-Transitions technique [2] performs a binary search of the memory states of a program between a passing test case and a failing test case; this technique is part of a suite of techniques defined by Zeller and colleagues called *Delta Debugging.* The Cause-Transitions technique defines a method to automate the process of making hypotheses about how state changes will affect output. In this technique, the program under test is stopped in a symbolic debugger using a breakpoint—for both a passed test case and failed test case. Part of the memory state is swapped between the two runs and then allowed to continue running to termination. The memory that appears to cause the failure is narrowed down using a technique much like a binary search with iterative runs of the program in the symbolic debugger. This narrowing of the state is iteratively performed until the smallest state change that causes the original failure can be identified. This technique is repeated at several program points to find the flow of the differing states causing the failure throughout the lifetime of each run. These program points are then used as the initial set of points from which to search for the fault.

After this set of program points has been defined, they are specified as the initial set of statements that the programmer uses to search for the faults. If the fault is not contained in this initial set, they too prescribe the SDG-ranking technique to guide the programmer's efforts in finding the fault. They also specify two improvements to the SDG-ranking technique that can exploit the programmer's knowledge of whether particular states are "infected" by a fault or "causes" the fault to be manifest. These improvements are called *exploiting relevance* and *exploiting infections* and are defined in Reference [2].

3. EMPIRICAL STUDY

We conducted a study to compare Tarantula to the other four existing fault-localization techniques that were described in Section 2. This section describes our empirical study, and presents the results of the study and an analysis of it.

3.1 Variables and Measures

3.1.1 Independent Variables

Our experiment manipulated one independent variable: the fault-localization technique. The techniques that we examine are:

1. Set union
2. Set intersection
3. Nearest Neighbor
4. Cause Transitions
5. Tarantula

3.1.2 Dependent Variables and Measures

To compare these techniques, we use two dependent variables: effectiveness and efficiency. To evaluate the effectiveness of the techniques, we rank the statements of a program in terms of how the individual techniques specify their rankings. For the Set-union, Set-intersection, Nearest-Neighbor, and Cause-Transitions techniques, we use the SDG-ranking technique that is described in References [12] and [2]; we described this ranking technique in Section 2.2. These techniques produce an initial subset of program entities that are to be examined as suspicious. However, these subsets often exclude the fault. Thus, this ranking system specifies a way to order the remaining program entities in the search for the fault after the initial specified subsets are examined. For the Tarantula technique, we used the ranking system described in Section 2.1. This ranking system uses the "suspiciousness" scores to rank the executable statements in the program.

To evaluate the efficiency of the techniques, we recorded timings of using the Tarantula technique. The timings are

Table 1: Objects of Analysis

Program	Faulty Versions	Procedures	LOC	Test Cases	Description
print_tokens	7	20	472	4056	lexical analyzer
print_tokens2	10	21	399	4071	lexical analyzer
replace	32	21	512	5542	pattern replacement
schedule	9	18	292	2650	priority scheduler
schedule2	10	16	301	2680	priority scheduler
tcas	41	8	141	1578	altitude separation
tot_info	23	16	440	1054	information measure

gathered for both computational time and time required for necessary I/O. For the Cause-Transitions technique, we use the timing averages reported in [2]. For the other three techniques, we do not have recorded timings, but we can reliably estimate their efficiency relative to the two techniques for which we have recorded times. We discuss this in detail in Section 3.3.2.

3.2 Experiment Setup

3.2.1 Object of Analysis

For the object of analysis, we used the *Siemens suite* [3] of programs. We chose these programs because they are the most common object of analysis for comparing fault-localization techniques. The Siemens suite contains seven programs, faulty versions of those programs, and test suites designed to test those programs. Each faulty version contains exactly one fault, although the faults may span multiple statements or even functions. Table 1 provides a summary of the details about these subjects; see Hutchins et al. [3] for a more complete description of the programs, versions, and test suites. Combined, there are 132 faulty versions. Of these versions, we were able to use 122 versions. Two versions—versions 4 and 6 of print_tokens—contained no syntactic differences with the correct version of the program in the C file—there were only differences in a header file. In three other versions—version 10 of print_tokens, version 32 of replace, and version 9 of schedule2—no test cases fail, thus the fault was never manifested. In five versions—versions 27 and 32 of replace and versions 5, 6, and 9 of schedule—all failed test cases failed because of a segmentation fault. The instrumenter we used for our experiment (gcc with gcov) does not dump its coverage before the program crashes. Thus, we were unable to use these five versions for our study. After removing these ten versions, we have 122 versions for our studies. For comparison, Renieris and Reiss eliminated even more versions due to additional technical problems and used 109 versions for their studies.

3.2.2 Experiment Design and Analysis Strategy

For the Set-union, Set-intersection, and Nearest-Neighbor techniques, we use the results and implementation given in [12]. For the Cause-Transitions technique, we use the results and implementation given in [2].

For the Tarantula technique, we implemented the technique in Java. The object programs and versions are compiled with statement-level instrumentation using the GNU C compiler, gcc. The instrumented program is run, and the statement coverage is reported using the gcov program. The Tarantula tool reads in the coverage from gcov for each test case, parses its output, and represents the coverage in memory. Gcov reports the number of times each statement in the program was executed by each test case. The Tarantula tool represents any statements that are executed one or more times for a particular test case as simply "covered" and statements that are executed zero times as "uncovered." The statements that are executable and uncovered are distinguished from statements that are not executable—only those statements that are executable and uncovered are deemed "uncovered." Each executable statement is then given a suspiciousness score and then ranked according to the ranking system given in Section 2.1. The process of running the Tarantula tool on all versions is automated using shell scripts. All times for the efficiency study were measured on a 3 GHz Pentium PC.

To evaluate the effectiveness of the techniques, a score is assigned to every faulty version of each subject program. The score defines the percentage of the program that need not be examined to find a faulty statement in the program or a faulty node in the SDG. The ranking strategy for each technique is used to determine the rank number of the fault, and this rank number is used to compute the score. The Set-union, Set-intersection, Nearest-Neighbor, and Cause-Transitions techniques use the nodes of a system dependence graph (SDG) to determine the percentage of the program that must be examined. The Tarantula technique uses the subject program's source code. To be comparable with the SDG approach, we consider only executable statements to determine the score. This omits from consideration source code such as blank lines, comments, function and variable declarations, and function prototypes. We also join all multi-line statements into one source code line so that they will be counted only once. We do this to compare the techniques fairly—only statements that can be represented in the SDG are considered. Thus, the percentage of the program that need not be considered includes no unexecutable program entities, for *all* techniques in our experiment.

Faults and failures are identified by using the version of the subject programs that is deemed "correct." To identify the faults, the faulty version of the program is compared with the correct version. The lines in which they differ are recorded as the fault. To distinguish failing from passing test cases, we ran the correct version with each test case and recorded its output. We use these outputs to define the expected outputs for that program and test cases. We ran all faulty versions recording their outputs, and compared those with the expected output.

Table 2: Percentage of test runs at each score level.

Score	Tarantula	NN/perm	NN/binary	CT	CT/relevant	CT/infected	Intersection	Union
99-100%	13.93	0.00	0.00	4.65	5.43	4.55	0.00	1.83
90-99%	41.80	16.51	4.59	21.71	30.23	26.36	0.92	3.67
80-90%	5.74	9.17	8.26	11.63	6.20	10.91	0.00	0.92
70-80%	9.84	11.93	4.59	13.18	6.20	13.64	0.00	0.92
60-70%	8.20	13.76	3.67	1.55	9.30	4.55	0.00	0.00
50-60%	7.38	19.27	7.33	6.98	10.08	6.36	0.00	0.00
40-50%	0.82	3.67	9.17	3.10	3.88	1.82	0.00	0.00
30-40%	0.82	6.42	13.76	7.75	10.08	3.64	0.00	0.00
20-30%	4.10	1.83	13.76	4.65	3.10	7.27	0.00	0.00
10-20%	7.38	0.00	6.42	6.98	10.85	0.00	0.00	0.00
0-10%	0.00	17.43	28.44	17.83	4.65	20.91	99.08	92.66

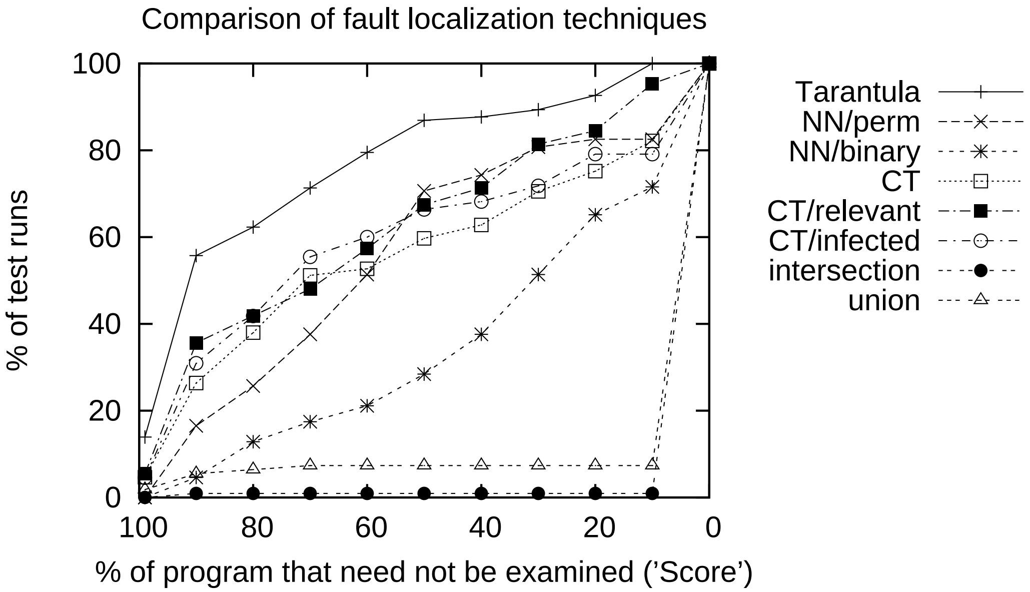

Figure 2: Comparison of the effectiveness of each technique.

3.3 Results and Analysis

3.3.1 *Effectiveness*

We represent the results concerning the effectiveness dependent variable in Table 2 and Figure 2. In Table 2 we show the percentage of versions that achieve a score within each segment listed. Following the convention used by both [12] and [2], each segment is 10 percentage points, except for the 99-100% range and the 90-99% range. We report our findings on the same segments. Note that whereas a 100% score is impossible[2] for all techniques considered, the first segment from 99-100% effectively "pinpoints" the fault in the program.

For example, in Table 2, for about 14% of the faulty versions and their test suites, the Tarantula technique was able to guide the programmer to the fault by examining less than one percent (a score of 99% or higher) of the executable code. At the next score level, 90-99%, we can see in Table 2 that the Tarantula technique is able to guide the programmer to the fault by examining less than 10% of the program for an additional 42% of the faulty versions and their test suites.

The results shown in Figure 2 depict the data in Table 2. Points and connecting lines are drawn for each technique. The legend to the right shows how to interpret the lines representing each technique. The labels in the legend are abbreviated for space. "NN/perm" is the Nearest-Neighbor technique using permutation distancing. "NN/binary" is the Nearest-Neighbor technique using binary distancing. "CT" is the Cause-Transitions technique using the standard SDG-ranking technique. "CT/relevant" is the Cause-Transitions technique exploiting relevance in the ranking technique. "CT/infected" is the Cause-Transitions technique exploiting infections in the ranking technique.

The horizontal axis represents the score measure defined above, which represents the percentage of the subject program that would not need to be examined when following the order of program points specified by the techniques. The vertical axis represents the percentage of test runs that are found at the score given on the horizontal axis. For the

[2] The best-case scenario is where the first ordered location is the fault. For this, the score is $(1-(1/\text{size of the program}))*100$.

Tarantula technique there is one test suite used for each faulty version, so the horizontal axis represents not only the percentage of test runs, but also the percentage of versions. For the Set-intersection, Set-union, and Nearest-Neighbor techniques, multiple test cases are chosen for each version (recall that each is dependent on which single failed test that is used). For these the vertical axis represents the percentage of all version-test pairs.

At each segment level, points and lines are drawn to show the percentage of versions for which the fault is found at the lower bound of that segment range or higher. For example, using the Tarantula technique, for 55.7% of the faulty versions, the fault was found by examining less than 10% of the executable code, thus achieving a score of 90% or better.

Overall, Figure 2 shows that the Set-intersection techniques perform the worst, followed by Set-union, then Nearest-Neighbor using binary distancing, then Nearest-Neighbor using permutation distancing, then the Cause-Transitions using different ranking strategies (some that leverage programmer knowledge), and then the best result is achieved by the fully automatic Tarantula technique.

The results show that at the 99% score level, Tarantula was able to effectively pinpoint the fault. At this level of specificity, Tarantula performs three times better than the previously known best technique—Cause Transitions—on these subjects. These results also show that this trend continues for every ranking—Tarantula was consistently more effective at guiding the programmer to the fault.

Table 3: Average time expressed in seconds.

Program	Tarantula (computation only)	Tarantula (including I/O)	Cause Transitions
print_tokens	0.0040	68.96	2590.1
print_tokens2	0.0037	50.50	6556.5
replace	0.0063	75.90	3588.9
schedule	0.0032	30.07	1909.3
schedule2	0.0030	30.02	7741.2
tcas	0.0025	12.37	184.8
tot_info	0.0031	8.51	521.4

3.3.2 *Efficiency*

Table 3 summarizes the efficiency results for the study. For Tarantula, both computational time and the time including computation and I/O are shown. For example, the table shows that for the program schedule2, the Tarantula technique required 0.0032 seconds of computational time and about 30 seconds to read and parse the coverage information about the test cases. For this same program, Cause Transitions requires over two hours to complete its analysis.

Although we do not have timing information for the Set-union, Set-intersection, and Nearest-Neighbor techniques, because of the way in which the computation is performed, we expect that they will be very similar to those found with the Tarantula technique. In the Set-intersection and Set-union techniques, set operations are performed over the set of statements for all passing test cases and a single failed test case. In the Nearest-Neighbor technique, a distance score must be defined for every passing test case, then set operations are performed over the set of statements using two test cases. In these three techniques, the SDG for the program is then traversed until the fault is found. We expect the computational time for these techniques and Tarantula to be similarly small. Moreover, the time required by the Tarantula technique for computation is quite small–in the thousandths of a second—thus, comparable times for other techniques will be indistinguishable by humans. The I/O cost should also be similar for these techniques. Recall that the Nearest-Neighbor technique needs to read in all coverage information for all passing test cases to determine which passing test case will be chosen as the "nearest" one to the failing test case used. Similarly for the Set-union and Set-intersection techniques, coverage information for all passing test cases and one failing test case must be read.

It is worth mentioning that Tarantula's I/O time can be greatly reduced with a more compact representation of the coverage information. Currently, the tool is using the output of gcov, which stores every test case's coverage in a text file that contains the program's full source code. For each test case, a text file of this format must be read in and parsed to extract which statement were executed.

Nonetheless, the results show a difference of about two orders of magnitude between Tarantula and Cause Transitions, indicating that for these programs the Tarantula technique is not only significantly more effective, but also much more efficient.

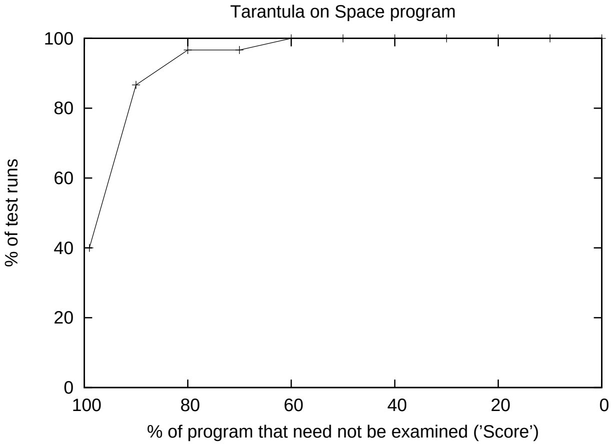

Figure 3: Results of the Tarantula technique on a larger program, Space.

3.4 Threats to Validity

There are a number of threats to the validity of this experiment. Specifically, the results obtained using the Siemens suite cannot be generalized to arbitrary programs. However, we expect that on larger programs with greater separation of concerns, all fault-localization techniques will do better. This expectation is supported by the results presented in our earlier paper [5] and the results summarized in Figure 3. In this figure, the Tarantula technique is applied to a program called Space containing 6218 lines of executable code, 38 faulty versions of Space, of which 30 versions produce failing test cases, and 13,585 test cases. On this larger subject program, Tarantula is much better at detecting the fault than on the smaller subjects. For 40% of the versions, the Tarantula technique guided the programmer to the fault by examining less than 1% of the code, effectively pinpointing the fault automatically. For 87% of the versions, the programmer needs to examine less than 10% of the program (score of 90% or higher) specified by Tarantula's ordering. We expect most fault-localization techniques to perform better on such larger programs, and would expect to see even better results on even larger programs that have an even greater separation of concerns.

The results presented in this experiment apply only to the case where the subjects used in the study each contain a single fault. We cannot generalize these results to these or any programs that have multiple faults. However, our previous studies [5], with a program containing multiple faults, suggests that the techniques can help to identify faults. In these studies, we evaluated versions of Space with up to five faults, and found that our technique could identify at least one fault in these multiple-fault versions. Furthermore, as faults are discovered and removed, others reveal themselves, which suggests an iterative process of using the technique.

In all techniques presented here, we assume that a programmer can identify the fault by inspecting the code—that is, she can follow the order of nodes or statements that is specified and determine at each one whether it is faulty. This applies further to the ranking modifications of the Cause-Transitions technique using the identification of infections. This issue must be explored further with human studies.

Another limitation to the experiment is that we did not implement all of the techniques evaluated here. This could be a factor when considering the efficiency results. Whereas the particular implementation may affect the efficiency of the techniques, the differences in timing results that we report are drastic enough (two orders of magnitude in some cases; see Table 3) that the implementation cannot explain these differences. Moreover, our technique is not optimized—it runs in a somewhat brute-force way, and it runs in an interpreted environment (Java).

4. DISCUSSION, CONCLUSIONS, AND FUTURE WORK

This paper presented the first set of empirical studies that compare four previously compared existing automatic fault-localization techniques—Set union, Set intersection, Nearest Neighbor, and Cause Transitions—with the Tarantula technique. The studies compare the five techniques in terms of effectiveness in focusing the programmer's attention on the likely faulty statements, and thus helping with the search for the fault. The results of the study are given in terms of the percentage of the program that need not be examined to locate the fault. The study shows that Tarantula consistently outperforms the other four approaches for the set of subjects studied. At the 99%-score level, Tarantula can pinpoint the fault three times more often than the technique previously shown to be the most effective at fault detection on this set of subjects. At the 90%-score level, Tarantula performs 57% better than this previous technique. The studies also compare the five techniques in terms of efficiency. Tarantula performs its computational analysis six orders of magnitude faster than the previously most effective technique and two orders of magnitude faster when considering I/O.

The studies also show that the Set-union, Set-intersection, and Nearest Neighbor techniques are less effective than the other two, especially at the higher scores. There are several possible causes for these differences:

- *Sensitivity.* The Set-union, Set-intersection, and Nearest-Neighbor techniques may be less effective because of the techniques' sensitivity to the particular test suites. This sensitivity was demonstrated on the example in Section 2. For the Nearest-Neighbor technique, removing the set of statements executed by the passed test case with the most similar coverage from the set of statements executed by the failed test case may cause the fault to be removed from the initial set in many cases. For this reason, we have designed our Tarantula technique to allow tolerance for passed test cases that occasionally execute faults.

- *Ranking technique.* The use of a breadth-first search over the SDG may not be an efficient strategy for exploring the program. The size of the set of nodes at each distinct distance ("rank") would likely grow quickly with the distance from the initial set up to a certain distance (beyond that distance, the size of the set of nodes would likely decrease rapidly). For this reason, we believe that it is important to provide an ordering of program points to guide the programmer from the program points that are most likely to be faulty to those least likely to be faulty (according to some approximation measures).

- *Use of single failed test case.* We have found that our Tarantula technique performs better with more failed test cases as well as more passed test cases. With our technique, we can observe its results with any subset of the test suite as long as it has at least one passed test case and at least one failed test case. We have found that utilizing the information from multiple failed test cases lets the technique leverage the richer information base. We believe that other fault-localization techniques would benefit by using multiple failed test cases as well.

In future work, we will further investigate these possible causes for differences.

The paper presented an overview of the techniques included in the studies (in Section 2). There are, however, several other fault-localization techniques that are related to the ones compared in the paper. Liblit and colleagues [7, 8] present a technique to monitor and locate faults in programs after they have been deployed. This technique uses sampling information—they instrument the program with probes that are randomly fired, and the sampled data is sent from the users' machines to the developer's site. The data also indicates whether the program terminated normally or with a segmentation fault. These data are then accumulated from all users and analyzed to attempt to find the source of the program crashing in the field. Their technique monitors branch coverage, return values, and invariant information. They represent each of these types of data as predicates. The predicates are then pruned using techniques much like those presented in the Section 2.2. In later work [8], they utilize equations similar to those used by the Tarantula tool to prune and rank the predicates in the program for inspection by the programmer. Their latest equations also exhibit a tolerance for faults that are occasionally executed by passed test cases. We did not include their technique in our study because their presented technique provides no way to quantify the ranking for all program points. They define an ordering for predicates, but if the fault lies outside a predicate, there is no technique presented to order these program points. In future work, we plan to identify a method for comparing their technique to the five presented here, and perform studies using this method.

Another area of related work is that of Ruthruff, Burnett, and Rothermel [13]. Ruthruff and colleagues present fault localization techniques for end-user programmers. Such end-user programming environments include spreadsheets in which programming is done by users with little or no formal programming education. Like Tarantula, they define a visualization for their techniques to enable the user to identify likely faulty areas. In recent work [13], they presented an empirical study that investigates the impact of the information from which they work and separately investigates the mapping to the interface for this information for the user. This approach could be useful in evaluating how our technique is both calculated and then displayed to the user, and we plan to consider such an evaluation for Tarantula. They also found that end-users often make mistakes in determining when some values are erroneous and thus prescribe techniques that have some tolerance for some incorrect interaction with the user. This technique is closely related to Tarantula's approach that also tolerates test cases that execute the fault but still produces the expected output.

There are a number of other directions for future work that we would like to pursue. Although we presented our technique here as a simple ranking for the purposes of demonstrating that Tarantula can be compared to other techniques using quantitative measures, we believe there is value in presenting a visualization that lets programmers understand the ranks and suspiciousness values that are computed (instead of simply reporting them). Programmers may be able to see that program points identified as suspicious are related in some way that might not be obvious from such a report. This role of the visualization in helping the programmer to locate faults has not yet been evaluated on real users, and we believe that this is an important area for future work.

Finally, to address some of the threats to validity, we plan to perform more studies to evaluate the way Tarantula and the other techniques perform on other subjects, such as larger subjects and subjects containing more than one fault. Although the Siemens' suite is the common suite of programs on which several researchers have evaluated their work, programs of greater size and number of faults may provide greater ability to generalize the results.

5. ACKNOWLEDGMENTS

This work was supported in part by National Science Foundation awards CCR-0096321, CCR-0205422, SBE-0123532 and EIA-0196145 to Georgia Tech, by Tata Consultancy Services, and by the State of Georgia to Georgia Tech under the Yamacraw Mission. The anonymous reviewers provided many helpful suggestions to improve the paper. Eli Tilevich also provided many suggestions to improve the paper.

6. REFERENCES

[1] 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.

[2] 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.

[3] 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, Sorrento, Italy, May 1994.

[4] J. Jones, M. J. Harrold, and J. Stasko. Visualization for fault localization. In *Proceedings of the Workshop on Software Visualization, 23rd International Conference on Software Engineering*, Toronto, Ontario, May 2001.

[5] 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.

[6] J. Jones, A. Orso, and M. Harrold. Gammatella: Visualizing program-execution data for deployed software. *Palgrave Macmillan Information Visualization*, 3(3):173–188, Autumn 2004.

[7] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the Conference on Programming Language Design and Implementation*, San Diego, California, June 2003.

[8] 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.

[9] A. Orso, J. Jones, and M. J. Harrold. Visualization of program-execution data for deployed software. In *Proceedings of the ACM Symposium on Software Visualization*, pages 67–76, San Diego, California, June 2003.

[10] H. Pan, R. A. DeMillo, and E. H. Spafford. Failure and fault analysis for software debugging. In *Proceedings of COMPSAC 97*, pages 515–521, Washington, D.C., August 1997.

[11] H. Pan and E. Spafford. Heuristics for automatic localization of software faults. Technical Report SERC-TR-116-P, Purdue University, 1992.

[12] 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.

[13] J. R. Ruthruff, M. Burnett, and G. Rothermel. An empirical study of fault localization for end-user programmers. In *Proceedings of the International Conference on Software Engineering*, pages 352–361, St. Louis, Missouri, May 2005.

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

[15] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, pages 1–10, Charleston, South Carolina, November 2002.