SEMI-AUTOMATIC FAULT LOCALIZATION

A Dissertation
Presented to
The Academic Faculty

by

James Arthur Jones

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
April 2008

SEMI-AUTOMATIC FAULT LOCALIZATION

Approved by:

Mary Jean Harrold, Committee Chair
School of Computer Science
Georgia Institute of Technology

Steven Reiss
Department of Computer Science
Brown University

Alessandro Orso
School of Computer Science
Georgia Institute of Technology

Spencer Rugaber
School of Computer Science
Georgia Institute of Technology

Santosh Pande
School of Computer Science
Georgia Institute of Technology

Date Approved: November 19, 2007

To the light that resides in us all, and to those that choose to see and nurture it in each other.

ACKNOWLEDGEMENTS

I have long dreamed of earning a doctorate degree, and this dream would not have come true if I had not had an incredible amount of support and encouragement throughout my life. I have been blessed to have been surrounded by extraordinary family, friends, and colleagues whose kindness and support have been instrumental.

First, I would like to thank my parents who have provided my foundation of support, in every way, from the very beginning. They have both encouraged me to pursue my passions and allowed me the space to figure what those passions were. They have offered the perfect amount of support: offering guidance when I needed it, but also, stepping back when I needed to experience some tough lessons on my own. I did. My father, Arthur Curtis Jones, has instilled my curiosity and love of the technical and has provided a grounding of support and pragmatic advice. My mother, Jennifer Lynn Jones, leads by example in demonstrating strength and grace. Her undying love and support have been a true inspiration. I want to thank my parents for making this and all my successes possible.

I owe an enormous debt of gratitude to my advisor and mentor, Dr. Mary Jean Harrold. She has worked tirelessly for my successes: far beyond the expectations of an advisor. I have never before witnessed someone so committed to her students and their successes. At the time of this writing, it has been over eleven years since she first invited me to participate as an undergraduate in her research group. In this time, I have come to view her as a mentor, role model, and dear friend. I want to thank Mary Jean from the bottom of my heart for believing in me, giving me a chance, showing me that I was capable of achieving this dream, and for providing the means and wisdom to achieve it.

I would like to thank my wife and love, Melissa Renee Jones. Her passion and drive inspire me to strive for "only the best" for us both. Throughout our years together, my Ph.D. work has at times been difficult and at times been grand. In the difficult times, she strengthened me and lifted me up with encouraging words and reasoned advice. In the grand times, she showed me how to let loose and enjoy the moment. I want to thank Melissa for her efforts to help me see that I deserve success in achieving my dreams.

Over my years as a Ph.D. student, I have had the pleasure of working with a number of friends that have been supportive. Dr. Alessandro Orso is a dear friend that has offered advice and support throughout my years as a Ph.D. student. Dr. James Bowring is a great friend who has offered advice, support, and an example of how it is done. I would like to thank my friends, current and former members of the Aristotle Research Group. This group has provided me a safe and stimulating environment to become a researcher and scientist.

Finally, I would like to thank my committee: Mary Jean Harrold, Alessandro Orso, Santosh Pande, Steven Reiss, and Spencer Rugaber. Their time and care in reading, and their advice, have greatly improved this dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

One of the most expensive and time-consuming components of the debugging process is locating the errors or faults. To locate faults, developers must identify statements involved in failures and select suspicious statements that might contain faults. In practice, this localization is done by developers in a tedious and manual way, using only a single execution, targeting only one fault, and having a limited perspective into a large search space.

The thesis of this research is that fault localization can be partially automated with the use of commonly available dynamic information gathered from test-case executions in a way that is effective, efficient, tolerant of test cases that pass but also execute the fault, and scalable to large programs that potentially contain multiple faults. The overall goal of this research is to develop effective and efficient fault localization techniques that scale to programs of large size and with multiple faults. There are three principle steps performed to reach this goal: (1) Develop practical techniques for locating suspicious regions in a program; (2) Develop techniques to partition test suites into smaller, specialized test suites to target specific faults; and (3) Evaluate the usefulness and cost of these techniques.

In this dissertation, the difficulties and limitations of previous work in the area of fault-localization are investigated. These investigations informed the development of a new technique, called Tarantula, that addresses some key limitations of prior work in the area, namely effectiveness, efficiency, and practicality. Empirical evaluation of the Tarantula technique shows that it is efficient and effective for many faults. The evaluation also demonstrates that the Tarantula technique can loose effectiveness

as the number of faults increases. To address the loss of effectiveness for programs with multiple faults, supporting techniques have been developed and are presented. The empirical evaluation of these supporting techniques demonstrates that they can enable effective fault localization in the presence of multiple faults. A new mode of debugging, called parallel debugging, is developed and empirical evidence demonstrates that it can provide a savings in terms of both total expense and time to delivery. A prototype visualization is provided to display the fault-localization results as well as to provide a method to interact and explore those results. Lastly, a study on the effects of the composition of test suites on fault-localization is presented.

CHAPTER I

INTRODUCTION

Software errors significantly impact software productivity and quality, and the problem is getting worse. According to a study released in June 2002 by the Department of Commerce's National Institute of Standards and Technology (NIST), "Software bugs, or errors, are so prevalent and so detrimental that they cost the U.S. economy an estimated $59.5 billion annually, or about 0.6 percent of the gross national product." [53]

Attempts to reduce the number of delivered faults are estimated to consume between 50% and 80% of the software development and maintenance effort [19]. One of the most time-consuming, and thus expensive, tasks required to reduce the number of delivered faults in a program is *debugging*—the process by which errors discovered during testing are located and fixed. Published results of interviews conducted with experienced programmers [56] and the experience and informed judgement of my research group's industrial collaborators confirm that the task of locating the faults,[1] or *fault localization*, is the most difficult and time-consuming component of the debugging task (e.g., [73]). Because of this high cost, any improvement in the process of fault localization 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

[1] In this document, I use the words "faults" and "bugs" interchangeably.

backtrack until the faults are found. This process can be quite time-consuming.

There are a number of ways, however, that this approach can be improved. First, the manual process of identifying the locations of the faults can be very time consuming. A technique that can automate, or partially automate, the process can provide significant savings. Second, tools based on this approach lead developers to concentrate their attention locally instead of providing a global view of the software. An approach that provides a developer with a global view of the software, while still giving access to the local view, can provide more useful information. Third, the tools use results of only one execution of the program instead of using information provided by many executions of the program. A tool that provides information about many executions of the program can help the developer understand more complex relationships in the system. Also, by utilizing more executions, an approach can allow multiple faults to be found. This research addresses these limitations.

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. Many papers on debugging and fault-localization have been published in academic conferences and journals (e.g., [17, 19, 20, 33, 42, 43, 45, 47, 48, 56, 61, 75, 76]). Many of the techniques are based on an analysis, called *slicing*, developed by Weiser [75, 76] that, given a program point and a suspicious variable, determines all statements in the program that might affect the value of that variable at that point. These slicing-based techniques (e.g., [20, 29, 33, 45, 56]) result in a subset of the program that may contain the fault.

Although many fault-detection techniques have been developed, these techniques have limitations that impact their ability to produce results that scale to practical systems or generalize to fault-localization tasks. The first limitation concerns the scalability of the techniques themselves to real systems. The techniques compute a subset of the program statements in which the search for the fault should begin.

However, this subset can be quite large, and thus, the developer's inspection of this subset for the fault can require significant time. Furthermore, in some cases, the fault may not be contained in this subset. In these cases, the techniques offer no method for ordering or searching the remaining statements in the program. Finally, the techniques have not been evaluated on large programs that contain multiple faults. Thus, improvements to existing techniques or newly developed techniques, along with empirical evaluation of those techniques, are needed to provide automated fault-localization techniques that can be used in practice.

1.1 Thesis Statement

The thesis of this research is that fault localization can be partially automated with the use of commonly available dynamic information gathered from test-case executions in a way that is effective, efficient, tolerant of test cases that pass but also execute the fault, and scalable to large programs that potentially contain multiple faults.

1.2 Contribution

This proposed research will provide the following contributions to the software engineering community:

1. A technique to localize faults using commonly available testing information.

2. Techniques to manage test suites to better enable the fault localization of programs with multiple faults.

3. A technique to parallelize the debugging effort for programs containing multiple faults.

4. A cost model to evaluate the effectiveness of a parallelized approach to debugging.

5. Empirical demonstration of the practical use of these techniques.

6. A visualization that can be applied to fault-localization techniques.

CHAPTER II

BACKGROUND

This research builds on work in debugging. This section provides background material and describes previous research on fault localization and debugging.

2.1 Definitions

Throughout this document, certain terminology will be used repeatedly. I present some definitions here to provide a basis for the following work. IEEE provides a "standard glossary of software engineering terminology" [39] that can be used to define a number of terms that will used in this document. IEEE defines the terms "mistake," "fault," "error," and "failure" as such:

mistake:

> A human action that produces an incorrect result.

fault:

> (1) A defect in a hardware device or component; for example, a short circuit or broken wire. (2) An incorrect step, process, or data definition in a computer program.

error:

> The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. For example, a difference of 30 meters between a computed result and the correct result.

failure:

> The inability of a system or component to perform its required functions within specified performance requirements.

So, a person can make a mistake that can cause a fault in a program. This fault in the program may produce a failure in the result. The difference between the failure output and the expected output is the error. In fact, the IEEE standard notes [39]:

> The fault tolerance discipline distinguishes between a human action (a mistake), its manifestation (a hardware or software fault), the result of the fault (a failure), and the amount by which the result is incorrect (the error).

A *passed* test case is one that produces the expected output. A *failed* test case is a test case that does not produce the expected output. Furthermore, when a failure, or failed test case, is noticed, a software developer may attempt to fix the program by finding and fixing the fault. This process is called *debugging*. The term "to debug" is defined as [39]:

> To detect, locate, and correct faults in a computer program.

Thus, the debugging process can be decomposed into the process of (1) detecting faults by observing failures, (2) finding or locating the faults that are causing the failures, and (3) fixing those faults to eliminate the failures. This work addresses the second task, which is called *fault localization*.

2.2 Traditional Debugging Techniques

There are two approaches to finding bugs that are typically used by software developers.

The first technique is to place print statements in the program to cause the program to output additional information to be analyzed. A programmer identifies points in the program to get a glimpse of the runtime state. A common practice is to place

print statements to indicate that control has reached that particular point. Another common practice is to place print statements to output variable values. As the program is executed, the program generates the additional debugging output that can be inspected by the developer. There are a number of limitations of the use of print statements for debugging. The debugging output can be quite large. The placement of the print statements and the inspection of the output are both unstructured and ad hoc. Analysis and placement are typically based on intuition. Typically, the use of print statements for debugging utilizes only one of the failed test cases instead of utilizing the other test cases in the test suite.

Another common technique is the use of a symbolic debugger. A symbolic debugger is a computer program that is used to debug other programs. Symbolic debuggers support features such as breakpointing, single stepping, and state modifying. Breakpointing allows the programmer to stop the program at a particular program point to examine the current state. Single stepping allows the program to proceed to the next instruction after the current breakpoint and set the new breakpoint at that instruction. Many debuggers also allow the programmer to not only view the current state of a variable, but also to change its value and then continue execution. Symbolic debuggers are included with many development environments such as the Gnu C Compiler Toolkit, Eclipse, and Microsoft Visual Studio.

Typically, a developer will place breakpoints at places in the program that she feels are suspicious of being the bug. She will then inspect the state at this point. She can then single step through the program watching the state change at each execution of each statement. Examples of such symbolic debuggers are GDB [32], DBX [71], DDD [77], and those included with integrated development environments such as Eclipse [25] and Visual Studio [52]. The size of the state at each point in the program can be significant and there are many instances of statement executions that can be examined. Similar to the use of print statements, there is no guidance as

to where to focus the attention. All inspection and analysis is unstructured and ad hoc, and the analysis is typically based on intuition.

To help alleviate the difficulty of identifying where to place breakpoints, researchers have proposed techniques that provide the ability to be able to step backward in execution using a symbolic debugger. A common problem that developers face is placing a breakpoint and realizing that the execution had proceeded too far—the bug had already been executed. Instead of stopping the execution, setting a new breakpoint earlier, and re-executing, researchers proposed symbolic debuggers that are capable of backward-stepping (i.e., moving backward to a previous state of the program at a previous instruction). Balzer first proposed this functionality with the EXDAMS system for Fortran programs [9]. Agrawal, Demillo, and Spafford [34] also proposed a technique that allows a developer to move an execution backward to previous states. Their goal is to allow the developer to set breakpoints and work backward to determine the conditions that contributed to the failed execution. These techniques have traditionally suffered from a substantial execution overhead in terms of either execution time and/or in terms of the storage space needed to save all the necessary historical states of the program.

2.3 Algorithmic Debugging

Another group of techniques that have been proposed by researchers is called algorithmic debugging. These techniques decompose the problem of finding a bug by dissecting a complex computation. A complex computation is recursively decomposed to simpler subcomputations. Each of these subcomputations is checked for correctness. When the developer has determined that a subcomputation is incorrect, the fault can be localized. For example, if a computation is composed of two subcomputations which are both correct, but the parent computation is deemed incorrect, then it is the parent computation that contains the bug. In other words, it is the

composition of the subcomputations that is faulty.

Typically, algorithmic debugging focuses on logical programming languages such as Prolog. Shapiro [68] proposed the *Divide-and-Conquer* algorithm for debugging. The algorithm recursively searches the computation tree to localize the fault. Shapiro proved that if a computation is correct, then every subcomputation must also be correct. Consequently, if a program is incorrect, at least one subprogram computation must be incorrect. Extensions to imperative languages such as Pascal have been proposed by Renner [62]. In their work, subcomputations are mapped to the procedure level.

One limitation of these algorithmic debugging approaches is that an oracle must be provided for each computation and subcomputation. Having to provide such an oracle is often too expensive to be practical. Another limitation of the imperative approaches is that the precision is limited to the procedure or module level.

2.4 Knowledge-based Debugging

Another area of existing debugging techniques is knowledge-based debugging. These approaches are based on artificial-intelligence research and knowledge-engineering research. Knowledge-based debugging relies upon training knowledge of the intended behaviors of a system and knowledge of the usual types of failures. The training is performed manually by developers. As such, these approaches have been found not to be scalable to real-world programs and are limited to small example programs.

One knowledge-based technique is called PUDSY (for Program Understanding and Debugging SYstem) by Lukey[51]. This technique decomposes the program into a number of code fragments consisting of a small set of statements in the program. Each of these fragments is then compared against a knowledge base. This knowledge base is composed of several code fragments and the assertions that can be drawn from them. This knowledge base is built manually by the developer. Code chunks

that cannot be associated with a rule from the knowledge base may be symbolically evaluated. The composition of all of these rules for all of the fragments is evaluated to determine the overall assertions that can be drawn for the program. These assertions are then checked against the program specification. If the assertions match the program specification, the program is deemed correct. If the assertions do not match the program specification, the program is deemed to contain a bug. If this occurs, the program is then backtraced using the assertions to find the fault. There are a number of limitations to this approach. First, it is limited in size to very small programs. Second, the developer must have detailed specifications of the program in the same syntax as the assertions for the code fragments (or must be transformable to that syntax). Third, the developer must create a set of rules to put in the knowledge base. Finally, there is a high complexity of the symbolic evaluation of the program. In the paper, there is no evaluation of the time overhead required for this approach.

Another example technique is called for FALOSY (for FAult LOcalization SYstem) by Sedlmeyer and colleagues[67]. Like the PUDSY approach, in this approach the knowledge base is informed manually by the developer. The knowledge base associates output failure symptoms with fault-localization hypotheses. The developer identifies symptoms of erroneous output and associates those with places in the program that likely cause those types of symptoms. This approach attempts to do what developers do naturally by inferring from the output the places in the program that may be responsible for such failures. After a sufficient training period, the knowledge base should contain enough rules to make some attempts to create automatic hypotheses of the faults that cause failures. When a failure is found, the output for that failure is compared against the knowledge base to find the closest matching rule. Using this rule, a hypothesis is offered to the developer as to the location of the fault that caused that failure. This technique has some limitations. The developer is responsible for creating the inference rules from the failure output symptoms to the faults, which

can be time consuming. Another limitation is that this technique relies upon past failures' symptoms and diagnoses to predict future symptoms and diagnoses. There are no empirical studies that show this to be true in general.

2.5 Slicing-based Techniques

Another class of proposed debugging techniques uses program slices. Weiser proposed slicing [75, 76] as a way to isolate the part of the program that was responsible for a value at a particular location in a program. A *slice* is the set of program locations that may influence the value of a variable at a specific program location. The computation of the slice uses static-analysis techniques—control-flow and data-flow. Typically, a particular output-inducing statement is identified as one that produced a manifestation of the failure. A variable that is used at this statement is also identified. The statement and the variable, together, form the *slicing criterion*. The slice is calculated for the slicing criterion to determine all statements that could have influenced that point in any execution. The set of program points that are identified as the result of the slice is a reduced search space for the fault. Given that the output statement and variable actually produced an incorrect output, the fault must reside in the slice.

Researchers have developed extensions to slicing that utilize additional information. For example, Korel and Laski present a technique called *dynamic slicing* [45] that uses a test case to determine the set of statements that actually affected the value of the suspicious variable. Several applications and extensions of dynamic slicing have been proposed for fault localization (e.g., [20, 29, 33, 56]). Pan and colleagues present a set of dynamic-slice-based heuristics that use set algebra of test cases' dynamic slices for similar purposes [55]. These slicing-based techniques result in a subset of the program that may contain the fault.

2.5.1 Execution Slice-based Techniques

Realizing that the precise definition of the suspicious variable can be difficult to determine and that dynamic slices can be expensive to compute, researchers proposed another set of techniques that makes use of an *execution slice*—the set of statements that are executed by a program for a particular test case [2].

Several researchers have used coverage-based information for fault localization. Collofello and Cousins first presented a technique that uses information that is similar to execution slices—the statements between any two predicates in a program [18]. Agrawal and colleagues present a technique that computes the set difference of the statements covered by two test cases—one passed and one failed [3]. 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.

Some simple and common techniques described in Reference [61] for computing a subset of all of coverage entities[1] to use as a reduced search space for the fault 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 passed test cases P containing individual passed test cases p_i, and a single failed 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

$$E_{initial} = E_f - \bigcup_{p \in P} E_p \tag{1}$$

The intuition of the Set-union approach is that the fault is likely in the set of entities that are executed exclusively by the failed test cases. This intuition implies that every

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

time that the fault was executed it caused a failure.

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 failed test case. A set of statements is obtained by performing the intersection of the sets of statements for all passed test cases, and removing the set of statements executed by the failed test case. Informally, the technique results in the set of statements that are executed in every passed test case, but not in the failed execution. Using the same notation as Equation 1, the Set-intersection technique can be expressed as

$$E_{initial} = \bigcap_{p \in P} E_p - E_f \tag{2}$$

The intuition of the Set-intersection approach is that the failed test case missed executing some part of the program, and that this omission may be responsible for causing its failed status. Although the fault should not be in the omitted section, it is surmised that the fault may be highly related to it—either in terms of location in the code listing or in terms of control or data dependencies.

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.

Program *mid()* in Figure 1 inputs three integers 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.

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

		Test Cases					
		t1	t2	t3	t4	t5	t6
	`mid() {` `  int x,y,z,m;`	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
1:	`read("Enter 3 numbers:",x,y,z);`	●	●	●	●	●	●
2:	`m = z;`	●	●	●	●	●	●
3:	`if (y<z)`	●	●	●	●	●	●
4:	`  if (x<y)`	●	●			●	●
5:	`  m = y;`		●				
6:	`  else if (x<z)`	●				●	●
7:	`  m = y;  // *** bug ***`	●					●
8:	`else`			●	●		
9:	`  if (x>y)`			●	●		
10:	`  m = y;`			●			
11:	`  else if (x>z)`				●		
12:	`  m = x;`						
13:	`print("Middle number is:",m);`	●	●	●	●	●	●
	`}` Pass/Fail Status	P	P	P	P	P	F

Figure 1: Example program and test suite to demonstrate techniques.

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 applying 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 the technique removes these statements from the the set of statements executed by the failed test case, the initial set contains only one program entity—statement 7. In this case, the Set-union technique would have identified the fault in the initial set. However, note the sensitivity of this technique to the particular test cases used—for many test suites, the initial set is either the null set or a set that 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 [60, 61]. 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 control-flow and data-flow 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.

Researchers have found [42, 61] 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. Researchers recognized these techniques' ineffectiveness when faults are executed by occasional passed test cases, and this recognition motivated techniques that allow some tolerance for these cases.

2.5.2 Nearest Neighbor Technique

Renieris and Reiss [61] address the issue of tolerance for an occasional passed test case executing a fault with their *Nearest-Neighbor Queries* 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 [3], 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. 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 passed and failed 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 2.5.1) 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 the example program, *mid()* and its test suite presented in Figure 1. In this test suite, only one failed test case exists, thus the technique chooses it as the 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 the technique removes the set of statements executed by the passed test case from the set of statements executed by the failed test case, the result is the null set as the 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). The technique finds that the fifth test case is the passed test case with the least distance. When the technique removes the set of statements executed by the fifth test case from the set of statements executed by the failed test case, the set containing only statement 7 is obtained. In this case, the Nearest-Neighbor Queries technique would have identified the fault in the initial set. However, notice that this technique is also sensitive to the particular test cases used.

2.6 Memory Modifying Techniques

Cleve and Zeller's Cause-Transitions technique [17] performs a binary search of the memory states of a program between a passed test case and a failed 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

each program point throughout the execution of the test cases to find the flow of the differing states causing the failure throughout the lifetime of each run. The program points that are associated with a transition in the state that caused the failure are saved. 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, "causes" the fault to be manifest, or are "irrelevant" to the fault.

There are a number of limitations to such an approach. The main limitation is that the technique is expensive. For each execution point in the program (every execution instance of each statement), the program must be run multiple times to cause the executions to breakpoint there and then recursively narrow the search for the state that causes the failure. Cleve and Zeller found that the approach required over two hours to complete for a program of about 300 lines of code. Also, the technique requires that two test cases—one passed and one failed—be found that have nearly identical execution paths through the program. Otherwise, the breakpoints cannot be placed throughout the execution. It may be difficult to find such test cases that are nearly identical in execution path, but produce different pass/fail statuses.

2.7 Techniques Extending Concepts Presented in Our Work

Since the initial publication of our work, others have proposed extensions to the concepts that we presented. I will describe some of these techniques in Section 3.6 after describing the fundamentals of our work.

CHAPTER III

FAULT LOCALIZATION USING TESTING INFORMATION

This chapter presents our technique that utilizes commonly available testing information in a way that is effective, efficient, tolerant of test cases that pass but also execute the fault, and scalable to large programs. This chapter first introduces the intuition and overall approach of the technique. The chapter then defines metrics that are used to support the technique and a ranking technique that is used to evaluate it. The chapter next provides an analysis of the technique. Finally, the chapter presents some related work that has extended some of the concepts of our fault-localization technique since it was first presented.

3.1 General Technique

Software testers often gather large amounts of data about a software system under test. These data, such as coverage, 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. Our technique uses information provided by these data for our fault-localization technique called *Tarantula*.

Tarantula utilizes 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 choice to use information sources that are commonly available in practice was a deliberate one. Many organizations already use tools that enable dynamic instrumentation of the program to determine its test suite coverage adequacy.

In fact many commonplace tools support coverage instrumentation such as the GNU C compiler (gcc and gcov) [31], jcoverage (Eclipse plug-in) [40], InsECTJ (Eclipse plug-in) [16], Cobertura [23], Rational PureCoverage [38], and BullseyeCoverage (Visual Studio plug-in) [15].

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 use coverage information (e.g., [3, 61]), Tarantula permits the fault to be occasionally executed by passed test cases. We have found that this tolerance often provides for more effective fault localization.

At a high-level, the technique assigns two metrics to every coverage entity (e.g., statements, branches, methods) being monitored, and uses the values of these metrics for each coverage entity to rank the entities. The software developer is then directed to focus his or her attention to the highest ranked entities when searching for the fault. The next section discusses why these metrics need to have a tolerance for passed test cases that execute the fault. Section 3.3 defines the metrics. Section 3.4 defines how the entities are ranked using these metrics.

3.2 Faults Executed by Passed Test Cases

We found in practice and in our experiments that faults were often executed by a few passed test cases. Most existing slicing-based techniques would remove the fault from the area of the program that they deem suspicious of being the fault. The assumption that these techniques make is that every time a fault is executed, it must cause a failure. However, we found that this assumption doesn't hold, and it is a major source of the ineffectiveness of the existing slicing-based techniques. Renieris and Reiss report such losses of effectiveness due to removing the true fault from the set of suspected faulty statements [61, p. 35]. They report:

> While collecting traces, we observed that in some cases, spectra of successful and failed runs collided. That is, the spectra of some failed runs were indistinguishable from the spectra of some successful runs for the same version of the program.

They found that the entities that were executed by the failed and passed runs were sometimes the same. Thus, in these cases the fault must have been executed by at least one passed run.

The observation that faults can occasionally be executed in a passed context has motivated our approach that has some tolerance for this phenomenon. By relaxing the condition of suspiciousness from "only executed by failed test cases" to "primarily executed by failed test cases," the fault localization becomes much more resilient to these situations.

3.3 Metrics

The Tarantula technique computes and assigns two metrics to the coverage entities that are being considered. These metrics are called *suspiciousness* and *confidence*. The suspiciousness metric represents, for its corresponding coverage entity, a level of suspicion of being a fault that caused the failed test cases in the test suite to fail. The value of the suspiciousness metric ranges from 0 to 1. A suspiciousness value of 0 represents an entity that is least suspicious of causing the failed test cases. A suspiciousness value of 1 represents an entity that is most suspicious of causing the failed test cases. Between these two extremes is a continuous range of values that represents relative values of suspicion that can be assigned to the coverage entities. Given a test suite T, the suspiciousness metric for an entity e is defined as:

$$suspiciousness(e) = \frac{\frac{failed(e)}{totalfailed}}{\frac{passed(e)}{totalpassed} + \frac{failed(e)}{totalfailed}} = \frac{\%failed(e)}{\%passed(e) + \%failed(e)} \qquad (3.3.1)$$

In Equation 3.3.1, $failed(e)$ is the number of failed test cases in T that executed statement s one or more times. Similarly, $passed(e)$ is the number of passed test cases in T that executed entity e one or more times. $totalpassed$ and $totalfailed$ are the total number of test cases in T that pass and fail, respectively. For these metrics, we use a division operator that evaluates to zero if the denominator is zero. Another way to interpret this equation is to express each the passed and failed ratios as percentages. Doing so may allow for an easier interpretation. In this representation, $\%passed(e) = \frac{passed(e)}{totalpassed} * 100$ and $\%failed(e) = \frac{failed(e)}{totalfailed} * 100$.

The confidence metric was defined that expresses the confidence in the suspiciousness value that was computed. The intuition is that the more execution information from either class (pass or fail) that is available, the more confident we can be in the suspiciousness value that is given. Like the suspiciousness metric, the confidence ranges from 0 to 1, inclusively. A coverage entity that has a confidence value of 0 is one for which we have no confidence in the suspiciousness value that is assigned to it. A coverage entity that has a confidence value of 1 is one for which we have a high level of confidence in the suspiciousness value that is assigned to it. Between these two extremes is a continuous range of values that represents relative values of confidence that can be assigned. The confidence metric for an entity e is defined as:

$$confidence(e) = max\left(\frac{passed(e)}{totalpassed}, \frac{failed(e)}{totalfailed}\right) = max\left(\frac{\%passed(e)}{100}, \frac{\%failed(e)}{100}\right) \tag{3.3.2}$$

In Equation 3.3.2, the variables are the same as those defined above for Equation 3.3.1.

Both the suspiciousness and the confidence metrics can be applied to various coverage entities. Some examples of such coverage entities are statements, branches, definition-use pairs, procedures, procedure calls, and variable-value ranges. Any type of program entity for which we can instrument to determine whether it was executed or

not executed by each test case can be used with these metrics. In fact, for a program that has a mapping from requirements to the source code, the technique can be applied to the requirements. For illustration, much of the text here will be describing the technique at the statement level. The statement level is a convenient and practical level of instrumentation for a number of reasons. First, many developers and testers already instrument their program at the statement level to determine the level of testing adequacy for their test suites. Second, many commonplace tools provide statement-level instrumentation, such as GNU gcc and many versions of Microsoft Visual Studio. Finally, a program instrumented for statement coverage has relatively low instrumentation overhead compared to other types of coverage, such as definition-use pairs or scalar-value-pair invariants.

Figures 2 and 3 present the algorithm that assigns the suspiciousness and confidence metrics. In Figure 2, Lines 2–10 count the numbers of passed and failed test cases. Lines 11-25 count the numbers of passed and failed test cases that execute each coverage entity. These lines also compute the suspiciousness and confidence values for each coverage entity by calling the functions listed in Figure 3. These metrics were defined in Equations 3.3.1 and 3.3.2.

To illustrate how the Tarantula technique works, consider the example program, *mid()*, and test suite given in Figure 4. Program *mid()*, described in Section 2, inputs three integers and outputs the median value. Recall that the program contains a fault on Statement 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 three columns labeled "suspiciousness," "confidence," and "rank." The suspiciousness column shows the suspiciousness score that the technique computes for each statement. The confidence column shows the confidence score for each statement. The last column shows a rank

Algorithm: AssignMetrics

Input : $M[C, T]$: a coverage matrix of boolean values specifying which test cases in T executed each coverage entity in C, where T is the list of test cases $[T_1, T_2, ..., T_m]$ and C is the list of coverage entities $[C_1, C_2, ..., C_n]$
P: a list of boolean values $[P_1, P_2, ..., P_m]$ specifying whether each test case in T passed

Output: S: a list of suspiciousness values $[S_1, S_2, ..., S_n]$ for each coverage entity in C
F: a list of confidence values $[F_1, F_2, ..., F_n]$ for each coverage entity in C

Declare: p_A: number of passed test cases in T
f_A: number of failed test cases in T
p_i: number of passed test cases that executed entity C_i in C
f_i: number of failed test cases that executed entity C_i in C

```
1  begin
2      p_A ← 0
3      f_A ← 0
4      foreach test case T_j in T do
5          if P_j then
6              p_A ← p_A + 1
7          else
8              f_A ← f_A + 1
9          end
10     end
11     foreach coverage entity C_i in C do
12         p_i ← 0
13         f_i ← 0
14         foreach test case T_j in T do
15             if M[C_i, T_j] then
16                 if P_j then
17                     p_i ← p_i + 1
18                 else
19                     f_i ← f_i + 1
20                 end
21             end
22         end
23         S_i ← Suspiciousness(p_A, f_A, p_i, f_i)
24         F_i ← Confidence(p_A, f_A, p_i, f_i)
25     end
26 end
```

Figure 2: `AssignMetrics` algorithm used for the Tarantula technique.

```
Function: Suspiciousness(p_A, f_A, p_i, f_i)
Input  : p_A: number of passed test cases
         f_A: number of failed test cases
         p_i: number of passed test cases that executed the considered entity
         f_i: number of failed test cases that executed the considered entity
Output : S: suspiciousness value
Declare: R_p: passed ratio
         R_f: failed ratio
1 begin
2 |  R_p ← Safe_divide (p_i, p_A)
3 |  R_f ← Safe_divide (f_i, f_A)
4 |  S ← Safe_divide (R_f, R_f + R_p)
5 end
```

```
Function: Confidence(p_A, f_A, p_i, f_i)
Input  : p_A: number of passed test cases
         f_A: number of failed test cases
         p_i: number of passed test cases that executed the considered entity
         f_i: number of failed test cases that executed the considered entity
Output : F: confidence value
1 begin
2 |  F ← max( Safe_divide(f_i, f_A), Safe_divide(p_i, p_A))
3 end
```

```
Function: Safe_divide(n, d)
Input  : n: numerator
         d: denominator
Output : r: result of the safe division
1 begin
2 |  if d = 0 then
3 |  |  r ← 0
4 |  else
5 |  |  r ← n/d
6 |  end
7 end
```

Figure 3: `Suspiciousness()` and `Confidence()` functions (along with utility function `Safe_divide()`) used in the `AssignMetrics` algorithm.

for each statement. This column will be discussed in Section 3.4

	Test Cases								
	t1	t2	t3	t4	t5	t6	suspiciousness	confidence	rank
`mid() { int x,y,z,m;`	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3			
`1: read("Enter 3 numbers:",x,y,z);`	●	●	●	●	●	●	0.5	1.0	7
`2: m = z;`	●	●	●	●	●	●	0.5	1.0	7
`3: if (y<z)`	●	●	●	●	●	●	0.5	1.0	7
`4: if (x<y)`	●	●			●	●	0.63	1.0	3
`5: m = y;`		●					0.0	0.2	12
`6: else if (x<z)`	●				●	●	0.71	1.0	2
`7: m = y; // *** bug ***`	●					●	0.83	1.0	1
`8: else`			●	●			0.0	0.4	9
`9: if (x>y)`			●	●			0.0	0.4	9
`10: m = y;`			●				0.0	0.2	12
`11: else if (x>z)`				●			0.0	0.2	12
`12: m = x;`							0.0	0.0	13
`13: print("Middle number is:",m);`	●	●	●	●	●	●	0.5	1.0	7
`}` Pass/Fail Status	P	P	P	P	P	F			

Figure 4: Example of Tarantula technique.

Consider Statement 1, which is executed by all six test cases and contains both passed and failed test cases. The Tarantula technique assigns Statement 1 a suspiciousness score of 0.5 because one failed test case out of a total of one failed test case in the test suite executes it (giving a ratio of 1), and five passed test cases out of a total of five passed test cases execute it in the test suite (giving a ratio of 1). Using the suspiciousness equation specified in Equation 3.3.1, the technique gets a suspiciousness value of $1/(1+1)$, or 0.5. Statement 1 has a confidence score of 1.0 because five of the five passed test cases executed it, and one out of one failed test cases executed it.

Consider also Statement 7, which is executed by one of the five passed test cases and by the one failed test case. Statement 7 has a suspiciousness value of 0.83, which

is more suspicious than Statement 1. Statement 7 is more suspicious than Statement 1 because it is executed primarily by the failed test cases—in this case, 100% of the failed test cases, but only 20% of the passed test cases.

3.4 Ranking of Program Entities

Using the *suspiciousness* and *confidence* scores, the technique sorts the coverage entities of the program under test. The sorted list of coverage entities gives a *ranking* of all entities. Each entity in the ranking has a *rank* that defines its depth in the list. The set of entities that have the greatest suspiciousness value is the set of entities to be considered first by the programmer when looking for the fault, and thus has the highest rank. 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.

The *confidence* score is used as a tie-breaker—high-confidence entities are ranked higher than lower ones. This specifies a ranking of entities in the program. For evaluation purposes, each set of entities at the same rank 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 consists of 10 statements, then every statement in that set is considered to have a rank of 10.

In Figure 4, the last column shows the rank of each statement according to the suspiciousness and confidence of the 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 by the suspiciousness, from the greatest score to the least score. Any statements that have the same suspiciousness score are further ordered based on the confidence score.

When Tarantula orders the statements according to suspiciousness, Statement 7 is the statement with the highest rank; then Statement 6; then Statement 4; then Statements 1, 2, 3, and 13 (at the same rank number); then Statements 8 and 9; then Statements 5, 10, and 11; and finally Statement 12 with the lowest rank. Statement 7 is the only statement in the highest rank, and thus, the programmer would inspect it first. If the fault were not at Statement 7, she would continue her search by looking at the statements at the next ranks. Note that the faulty statement 7 is ranked first—this means that programmer would find the fault at the first statement that she examined.

An evaluation of the Tarantula technique, compared to several other existing fault-localization techniques, is presented in an empirical study given in Chapter 7.

3.5 Analysis of the Suspiciousness Metric

This section provides a more detailed examination of the suspiciousness metric. First, the section presents the complexity analysis of the Tarantula fault-localization algorithm. Then, the section shows that the rankings provided by the suspiciousness metric are not biased toward either the number of passed test cases or the number of failed test cases that execute a statement—each affects the rankings equally. The section next describes a way to cast the Tarantula suspiciousness equation as a set similarity metric. Finally, the section describes a way to cast the use of the suspiciousness equation as a data-mining approach.

3.5.1 Complexity Analysis

The cost of the Tarantula algorithm is linear in the size of the program and linear in the size of the test suite. For each coverage entity in the program, the technique must count the number of test cases that executed it. To query each test case's coverage of an entity takes t steps where t is the number of test cases in the test suite. After the count of the number of passed and failed test cases that executed a coverage entity,

two calculations are performed to compute the suspiciousness and confidence scores. These computations are constant time operations—the sum of these constant time calculations is C. Thus, the time to compute the metrics for a single coverage entity is $t + C$. For all coverage entities, where the number of entities is n, the total time is $n(t + C)$. The run-time complexity is $O(tn)$.

3.5.2 Impartiality of Suspiciousness Equation

The suspiciousness equation provides a suspiciousness value to each program entity. These values can be used to assess the relative suspiciousness values for various entities and to enable a ranking of entities from greatest to least suspicious. The suspiciousness metric evaluates a program entity based on the degree to which it was involved in each category of test cases: passed and failed.

The suspiciousness equation serves as a ranking function to sort the entities. The suspiciousness equation provides a value ranging from 0 to 1, inclusively. The bounding of the range of values is useful for human comprehension. For example, a given statement can be said to be "100% suspicious." Another example of the usefulness of the bounding of the range is that it is easily mapped to a color space for visualization.

Although the suspiciousness equation bounds the range of suspiciousness values from 0 to 1, the ranking function that it provides is one that can be expressed in simpler terms. For the purpose of ranking, the suspiciousness is proportional to the number of failed test cases that executed that entity and inversely proportional to the number of passed test cases that executed it. The *totalpassed* and *totalfailed* values are the same for all statements in a program for a given test suite. With regard to the ordering or ranking of statements, the suspiciousness metric can be simplified to a simple ratio of *failed(s)* to *passed(s)* while still preserving its function as a ranking function. The suspiciousness equation is not biased for either the existence of failed or passed test cases.

We provide a direct proof that the suspiciousness equation computes a ranking function that is equivalent to the unbiased $\frac{failed(s)}{passed(s)}$ one by making a series of order-preserving transformations to the suspiciousness equation. For two arbitrary statements s_i and s_j, where $suspiciousness(s_i) > suspiciousness(s_j)$, We show that this implies that $\frac{failed(s_i)}{passed(s_i)} > \frac{failed(s_j)}{passed(s_j)}$.[1]

Theorem. For two arbitrary statements, s_i and s_j, such that $suspiciousness(s_i)$ is greater than $suspiciousness(s_j)$, the unbiased ratio of the number of failed test cases to number of the passed test cases that execute s_i is always greater than the unbiased ratio of the number of failed test cases to the number of passed test cases that execute s_j. This can be written as

$$suspiciousness(s_i) > suspiciousness(s_j) \implies \frac{failed(s_i)}{passed(s_i)} > \frac{failed(s_j)}{passed(s_j)} \quad (3.5.1)$$

Proof. We provide a direct proof of the theorem by deriving the consequent of the implication from the antecedent.

$$suspiciousness(s_i) > suspiciousness(s_j) \quad (3.5.2)$$

Inequality 3.5.2 shows that the *suspiciousness* value assigned to statement s_i is greater than the *suspiciousness* value assigned to statement s_j. By substituting the definition of the *suspiciousness* metric, we get

$$\frac{\%failed(s_i)}{\%passed(s_i) + \%failed(s_i)} > \frac{\%failed(s_j)}{\%passed(s_j) + \%failed(s_j)} \quad (3.5.3)$$

$$\frac{\frac{failed(s_i)}{totalfailed}}{\frac{passed(s_i)}{totalpassed} + \frac{failed(s_i)}{totalfailed}} > \frac{\frac{failed(s_j)}{totalfailed}}{\frac{passed(s_j)}{totalpassed} + \frac{failed(s_j)}{totalfailed}} \quad (3.5.4)$$

[1]The logic was inspired by a paper by Briand and colleagues [13].

We transform the fractions on each side of Inequality 3.5.4 to

$$\frac{failed(s_i) * totalpassed}{passed(s_i) * totalfailed + failed(s_i) * totalpassed} > \frac{failed(s_j) * totalpassed}{passed(s_j) * totalfailed + failed(s_j) * totalpassed} \tag{3.5.5}$$

We take the reciprocal of both sides of Inequality 3.5.5. Because $passed(s)$, $failed(s)$, $totalpassed$, and $totalfailed$ are always non-negative, the direction of the inequality operator is reversed to get

$$\frac{passed(s_i) * totalfailed + failed(s_i) * totalpassed}{failed(s_i) * totalpassed} < \frac{passed(s_j) * totalfailed + failed(s_j) * totalpassed}{failed(s_j) * totalpassed} \tag{3.5.6}$$

We simplify both sides of Inequality 3.5.6 to get

$$1 + \frac{totalfailed}{totalpassed} * \frac{passed(s_i)}{failed(s_i)} < 1 + \frac{totalfailed}{totalpassed} * \frac{passed(s_j)}{failed(s_j)} \tag{3.5.7}$$

We simplify Inequality 3.5.7 by adding (-1) to both sides and then multiplying both sides by $totalpassed/totalfailed$. Because the quantities $totalpassed$ and $totalfailed$ are always non-negative, the inequality operator remains the same. These quantities are not dependent on the individual statements, and thus for a given run of a test suite on a program, these values are constant for all statements. The result is

$$\frac{passed(s_i)}{failed(s_i)} < \frac{passed(s_j)}{failed(s_j)} \tag{3.5.8}$$

By taking the reciprocal of both sides of Inequality 3.5.8, we derive the consequent of the theorem's implication. Because the quantities $failed(s)$ and $passed(s)$ are always non-negative, the inequality operator is switched.

$$\frac{failed(s_i)}{passed(s_i)} > \frac{failed(s_j)}{passed(s_j)} \tag{3.5.9}$$

□

This series of order-preserving transformations demonstrates Implication 3.5.1. Thus, for the purposes of providing a ranking function, the unbiased ratio of the number of failed test cases that execute the entity to the number of passed test cases that execute the entity is equivalent to the *suspiciousness* metric.

3.5.3 Suspiciousness as Set-Similarity

Abreu and colleagues [1] propose that the Tarantula technique can be expressed as a set-similarity metric. They propose that performing fault-localization in a way such as that of Tarantula—where entity coverage is used to assess each entity's relative involvement in passed and failed test cases—is in essence a set-similarity problem. The insight of their analysis is to consider the set of test cases that execute each entity: one set per entity. The *reference set* to which each of these is compared is the set of test cases that fail. A similarity measure is calculated for each entity in the program. An entity that has a high similarity has a high suspiciousness of being a fault. Consequently, an entity that has a low similarity to the reference set has a low suspiciousness of being a fault that caused the failed test cases.

To demonstrate the key insight of their analysis, consider the example in Figure 4. The reference set contains only one test case: $t6$. The set for statement 1 is $\{t1, t2, t3, t4, t5, t6\}$. To calculate the suspiciousness of statement 1, the set $\{t6\}$ is compared to $\{t1, t2, t3, t4, t5, t6\}$. Intuitively, statement 7 has the most similar membership to the reference set.

In their terminology, $p \in \{0, 1\}$ indicates whether the entity s was executed (1 is executed, 0 is not-executed), $q \in \{0, 1\}$ indicates whether the test case passed or failed (1 is failed, 0 is passed), and a_{pq} represents the number of test cases that match

these two conditions. For example, $a_{11}(s)$ represents the number of failed test cases that executed entity s.

Using these definitions, the Tarantula suspiciousness equation can be expressed as:

$$suspiciousness(s) = \frac{\frac{a_{11}(s)}{a_{11}(s)+a_{01}(s)}}{\frac{a_{11}(s)}{a_{11}(s)+a_{01}(s)} + \frac{a_{10}(s)}{a_{10}(s)+a_{00}(s)}} \tag{3.5.10}$$

We can represent these values terms of the quantities that were defined in Section 3.3 as such:

$$a_{11}(s) = failed(s) \tag{3.5.11}$$

$$a_{10}(s) = passed(s) \tag{3.5.12}$$

$$a_{01}(s) = totalfailed - failed(s) \tag{3.5.13}$$

$$a_{00}(s) = totalpassed - passed(s) \tag{3.5.14}$$

Other set-similarity metrics could also be used for the purposes of fault localization. The Jaccard similarity coefficient and the Ochiai similarity coefficient are two metrics that are used for comparing the similarity and diversity of sample sets.

The Jaccard equation used in Reference [1] can be represented as:

$$\begin{aligned} suspiciousness_J(s) = & \quad \frac{a_{11}(s)}{a_{11}(s) + a_{01}(s) + a_{10}(s)} \\ = & \quad \frac{failed(s)}{totalfailed + passed(s)} \end{aligned} \tag{3.5.15}$$

The Ochiai equation used in Reference [1] can be represented as:

$$\begin{aligned} suspiciousness_O(s) = & \quad \frac{a_{11}(s)}{\sqrt{(a_{11}(s) + a_{01}(s)) * (a_{11}(s) + a_{10}(s))}} \\ = & \quad \frac{failed(s)}{\sqrt{totalfailed * (failed(s) + passed(s))}} \end{aligned} \tag{3.5.16}$$

These other similarity metrics can be used in place of the Tarantula one and also provide evidence toward my thesis statement.

3.5.4 Suspiciousness as Data-Mining

Denmat and colleagues [21] provided an analysis of the Tarantula technique that reinterprets the suspiciousness equation as a way to perform data-mining. In data-mining, association rules are defined among data, and the suspiciousness equation is one such association rule. Association rules seek to find hidden associations in large data sets. Agrawal and colleagues [4] describe finding association rules from analyzing supermarket sales data. They found rules such as "if a customer buys fish and lemon then he will probably also buy rice." Data-mining techniques must also be tolerant of cases where rules generally apply, but occasionally are violated. For example, there may be an occasional customer that bought fish and lemon, but did not buy rice. For the purposes of fault localization, the suspiciousness metric can be used to characterize rules that associate entity coverage with a failed result. Their work offers a formal justification of this work in a well-established area of computer-science research.

3.6 Techniques Extending Concepts Presented in Our Work

Since the publication of the Tarantula technique, others have proposed ways to use test-case coverage information to localize faults in a way that is also tolerant of passed test cases that execute the fault. This section describes two such techniques: Statistical Bug Isolation [48] and SOBER [49]. Each of these techniques further supports my thesis that fault-localization can be performed efficiently using commonly available testing information.

3.6.1 Statistical Bug Isolation

Liblit and colleagues [48] proposed a technique, called Statistical Bug Isolation (SBI)[2] for computing the suspiciousness of a predicate P, which they call *Failure*. With the assumption that the probability of P's being true implies failure, they compute the *Failure* of P by

$$Failure(P) = \frac{failed(P)}{passed(P) + failed(P)} \tag{3.6.1}$$

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. The predicate types that they evaluate are: branches in the code, error type return values from functions, and local scalar variable invariants.

They proposed their technique for use on deployed software. To minimize the runtime overhead of the instrumentation, they provided a statistical model that selectively and randomly executes the instrumentation probes in the code. Because they were not able to get full instrumentation information, they provided additional metrics to help account for the missing information. The other metrics that they provided were *Context* and *Increase*. The *Context* metric is defined as:

$$Context(P) = \frac{failed(P \vee \neg P)}{passed(P \vee \neg P) + failed(P \vee \neg P)} \tag{3.6.2}$$

and the *Increase* metric is defined as:

$$Increase(P) = Failure(P) - Context(P) \tag{3.6.3}$$

These additional metrics try to determine how much does P being true increase the probability of failure over simply sampling the predicate. All predicates that have a positive *Increase* are suspected of being faulty.

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

The main difference of SBI and Tarantula is that SBI targets localization of fault in deployed software. Liblit and colleagues have specialized the equations for statistical sampling of the coverage information to reduce the overhead of in-the-field instrumentation overhead. Tarantula targets in-house testing where full instrumentation is performed and complete coverage information can be gathered.

3.6.2 SOBER

Whereas Statistical Bug Isolation extended the Tarantula concepts to deployed software, Liu and colleagues extended the ideas to utilize branch profile counts instead of simple coverage hit vectors [49]. They called their technique SOBER. Liu and colleagues speculated that the data used for Tarantula and Statistical Bug Isolation was insufficient to capture the behavior of a test case. Tarantula and Statistical Bug Isolation only record whether an entity was *ever* covered during an execution. Liu and colleagues instead captured the number of times that an entity was executed for each execution. Using these entity profiles, they built statistical models of the behavior of both classes of test cases: passed and failed. These statistical models are used to evaluate each entity to determine if its execution counts discriminate between passed and failed contexts. Those that are found to be most different in these two contexts are considered to be the most suspicious.

The main difference of SOBER and Tarantula is that SOBER uses profiling information instead of coverage information. Their choice to use profiling information can increase the run-time instrumentation overhead and the fault-localization computation. We chose to use coverage information because it is our view that coverage information is more commonplace in current testing practice, and the use of commonly available dynamic information is a key goal of our research.

CHAPTER IV

HEURISTIC TECHNIQUE FOR PROGRAMS WITH MULTIPLE FAULTS

In practice, developers are aware of the number of failed test cases for their programs, but are unaware of whether a single fault or many faults caused those failures. Thus, developers usually target one fault at a time in their debugging. Programs that contain multiple faults present new challenges for fault localization. Most published fault-localization techniques target the problem of localizing a single fault in a program that contains only a single fault. In this chapter, I present our technique that locates multiple faults for programs that contain an arbitrary number of faults.

This chapter first describes the problem of interference that is caused by the presence of multiple faults. The chapter then presents two techniques that are used to solve this problem. The chapter next presents an analysis of each of these techniques. The chapter then describes two modes of debugging—sequential and parallel—that can be enabled by these solutions. Finally, the chapter presents related work.

4.1 The Interference of Multiple Faults

Fault-localization techniques, such as those described in Chapter 3, can be less effective for programs that contain multiple faults than for programs that contain only single, isolated faults. For example, consider a program and test suite for which a set of failed test cases, F_1, fail due to one fault, f_1, and another set of failed test cases, F_2, fail due to another fault, f_2. Fault f_1 may be executed only by the failed test cases in F_1, and fault f_2 may be executed only by the failed test cases in F_2. In such a scenario, each fault is not being primarily executed by all failed test cases. Thus,

fault-localization techniques that use the approach of identifying regions programs that are primarily executed by failed test cases may be less effective in such cases.

These fault-localization techniques attempt to find similarities among the failed test cases and then determine how these features differ from the passed test cases. In the case of programs that contain multiple faults, finding the similarities among the failed test cases that fail due to different faults may result in hypothesized fault locations that are unrelated to any of the faults. Execution traces for test cases that fail due to different fails may have similarities that do not help to distinguish the faults in the program—they may be due to chance or main-line code that all test cases must execute For these reasons, for any particular fault, the other faults create interference or *noise* that makes its localization by these types of fault-localization techniques more difficult. In our experimentation presented in Chapter 7, we provide evidence to support this claim.

The goal of my approach is to provide effective fault localization in the presence of multiple faults by the creation and specialization of test suites that target different faults. The key to this approach is the automatic partitioning of the failed test cases according to the faults that caused them. The approach creates subsets of the original test suite T that target individual faults. The approach partitions failed test cases into disjoint subsets of T. Each subset of failed test cases is called a *fault-focusing cluster*, and contains failed test cases that are similar in their execution behavior. Then, the approach creates *specialized test suites* by combining the passed test cases with each fault-focusing cluster. With these specialized test suites, the technique applies a fault-localization algorithm to automatically find the likely locations of the faults.

Consider the example shown in Figure 5, which is the same program that was shown in Figure 4, except that this program contains two faults. Statement 7 contains a fault: $m = y$ (the correct statement would be $m = x$). Statement 10 also contains

a fault: $m = z$ (the correct statement would be $m = y$). This modified example contains ten test cases, four of which are failed.

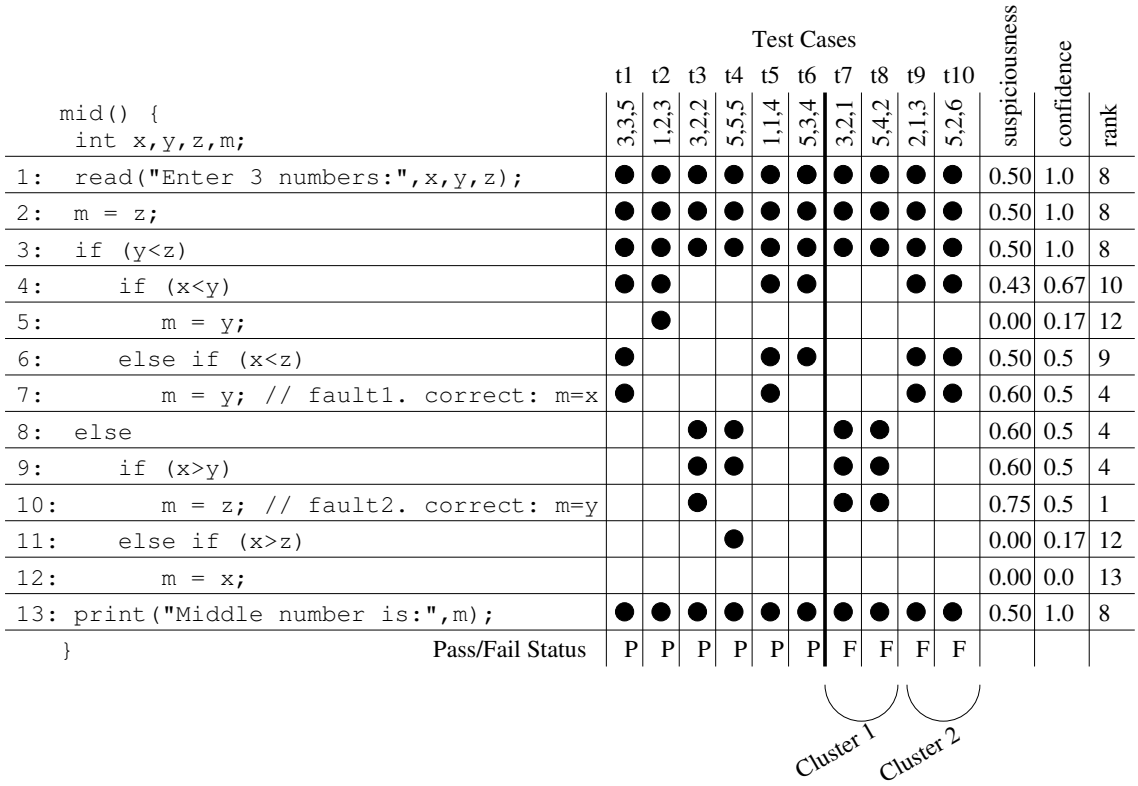

Figure 5: *mid()* and all test cases before any faults are located.

The four failed test cases are caused by the two different faults. The fault at Statement 10 causes test cases $t7$ and $t8$ to fail, and the fault at Statement 7 causes test cases $t9$ and $t10$ to fail. When applying the Tarantula technique to the entire test suite, all suspiciousness values and confidence values for the faulty statements are less than they could have been if there were only a single fault. The issue is that identifying the entities (in this case, statements) that were primarily executed by the failed test cases is more difficult when there are failures caused by multiple, different faults.

If we could know which failed test cases were failing due to each individual fault, we may be able to improve the effectiveness of the technique. However, knowing which

faults caused each failure would generally require knowledge of the faults. Because fault localization is an attempt to gather information about the nature of the fault, this is a recursive problem. Instead, we can group the failed test cases according to similar execution behavior, or fault-focusing clusters.

Returning to the example in Figure 5, it is clear to see that test cases $t7$ and $t8$ have similar execution behavior and $t9$ and $t10$ have similar execution behavior—in this case, the members of each group or cluster has exactly the same coverage vector. If we cluster these failed test cases into *Cluster 1* and *Cluster 2* (as shown at the bottom of the $t7$-$t10$ columns), we create two fault-focusing clusters. Combining each of these fault-focusing clusters with the passed test cases creates two specialized test suites: $\{t1, t2, t3, t4, t5, t6, t7, t8\}$ and $\{t1, t2, t3, t4, t5, t6, t9, t10\}$. These are shown in Figures 6 and 7, respectively.

	Test Cases										
	t1	t2	t3	t4	t5	t6	t7	t8	suspiciousness	confidence	rank
`mid() {` `  int x,y,z,m;`	3,3,5	1,2,3	3,2,2	5,5,5	1,1,4	5,3,4	3,2,1	5,4,2			
1: `read("Enter 3 numbers:",x,y,z);`	●	●	●	●	●	●	●	●	0.50	1.0	7
2: `m = z;`	●	●	●	●	●	●	●	●	0.50	1.0	7
3: `if (y<z)`	●	●	●	●	●	●	●	●	0.50	1.0	7
4: `  if (x<y)`	●	●			●	●			0.00	0.67	8
5: `  m = y;`		●							0.00	0.17	12
6: `  else if (x<z)`	●				●	●			0.00	0.5	9
7: `  m = y; // fault1. correct: m=x`	●				●				0.00	0.33	10
8: `else`			●	●			●	●	0.75	1.0	3
9: `  if (x>y)`			●	●			●	●	0.75	1.0	3
10: `  m = z; // fault2. correct: m=y`			●				●	●	0.86	1.0	1
11: `  else if (x>z)`				●					0.00	0.17	12
12: `  m = x;`									0.00	0.0	13
13: `print("Middle number is:",m);`	●	●	●	●	●	●	●	●	0.50	1.0	7
`}` Pass/Fail Status	P	P	P	P	P	P	F	F			

Figure 6: Example *mid()* with *Cluster 1*.

Figure 6 shows the results of the Tarantula technique for the specialized test suite

	Test Cases								suspiciousness	confidence	rank
	t1	t2	t3	t4	t5	t6	t9	t10			
`mid() {` `int x,y,z,m;`	3,3,5	1,2,3	3,2,2	5,5,5	1,1,4	5,3,4	2,1,3	5,2,6			
1: `read("Enter 3 numbers:",x,y,z);`	●	●	●	●	●	●	●	●	0.50	1.0	7
2: `m = z;`	●	●	●	●	●	●	●	●	0.50	1.0	7
3: `if (y<z)`	●	●	●	●	●	●	●	●	0.50	1.0	7
4: `if (x<y)`	●	●			●	●	●	●	0.60	1.0	3
5: `m = y;`		●							0.00	0.17	12
6: `else if (x<z)`	●				●	●	●	●	0.67	1.0	2
7: `m = y; // fault1. correct: m=x`	●				●		●	●	0.75	1.0	1
8: `else`			●	●					0.00	0.33	9
9: `if (x>y)`			●	●					0.00	0.33	9
10: `m = z; // fault2. correct: m=y`			●						0.00	0.17	12
11: `else if (x>z)`				●					0.00	0.17	12
12: `m = x;`									0.00	0.0	13
13: `print("Middle number is:",m);`	●	●	●	●	●	●	●	●	0.50	1.0	7
`}` Pass/Fail Status	P	P	P	P	P	P	F	F			

Figure 7: Example *mid()* with *Cluster 2*.

containing *Cluster 1* from Figure 5. Notice how the greatest suspiciousness value is assigned to the fault on Statement 10. Also, most of the confidence scores are greater for this specialized test suite than for the entire test suite.

Figure 7 shows the results of the Tarantula technique for the specialized test suite containing Cluster 2 from Figure 5. Unlike the use of the entire test suite, shown in Figure 5, and the other specialized test suite, shown in Figure 6, the use of this specialized test suite causes the fault localization technique to locate the fault on Statement 7.

4.2 Techniques for Clustering Failures

To achieve the goal of enabling a more effective fault-localization technique in the presence of multiple faults, we defined a debugging process that incorporates the

clustering of failed test cases. This process is shown by the dataflow diagram[1] in Figure 8. The program under test, P, is instrumented to produce $\hat{P}$. When $\hat{P}$ is executed, it produces execution information that is recorded, such as branch or method profiles. Executing $\hat{P}$ with test suite T results in some of the test cases being labeled as passed and the rest being labeled as failed. The passed test cases T_P and the failed test cases T_F are subsets of T. T_F and the execution information are input to the clustering technique, *Cluster*, to produce a set of fault-focused clusters C_1, C_2, ..., C_n that are disjoint subsets of T_F. Each C_i is combined with T_P to produce a specialized test suite that assists in locating a particular fault. Using these test suites, developers can debug each fault independently—shown as $Debug_i$ in the figure. When they find and fix the faults in the program, the resulting changes, ch_1, ch_2, ..., ch_n, are integrated into the program. This process can be repeated until all test cases pass.

The novel component of this debugging process, *Cluster*, is shown in more detail in Figure 9. We have developed two techniques to *Cluster* failed test cases. This section presents details of these techniques.

4.2.1 Clustering Based on Profiles and Fault-localization Results

The first fault-focused clustering technique, shown as Technique 1 in Figure 9, has three main components. The first component, *Behavior Model Clustering*, clusters behavior models of executions of failed test cases, T_F, to produce a complete clustering history (or dendrogram) D (described in Section 4.2.1.1). The second component, *Stopping-Criterion Calculation*, uses fault localization information to identify a stopping criterion for D, and produces a preliminary set of clusters, C_p (described in Section 4.2.1.2). The third component, *Refinement*, refines C_p by merging those clusters that appear to be focused on the same faults and outputs the final set of clusters,

[1]Rectangles represent processing components, edges represent the flow of data between the components, and labels on the edges represent the data.

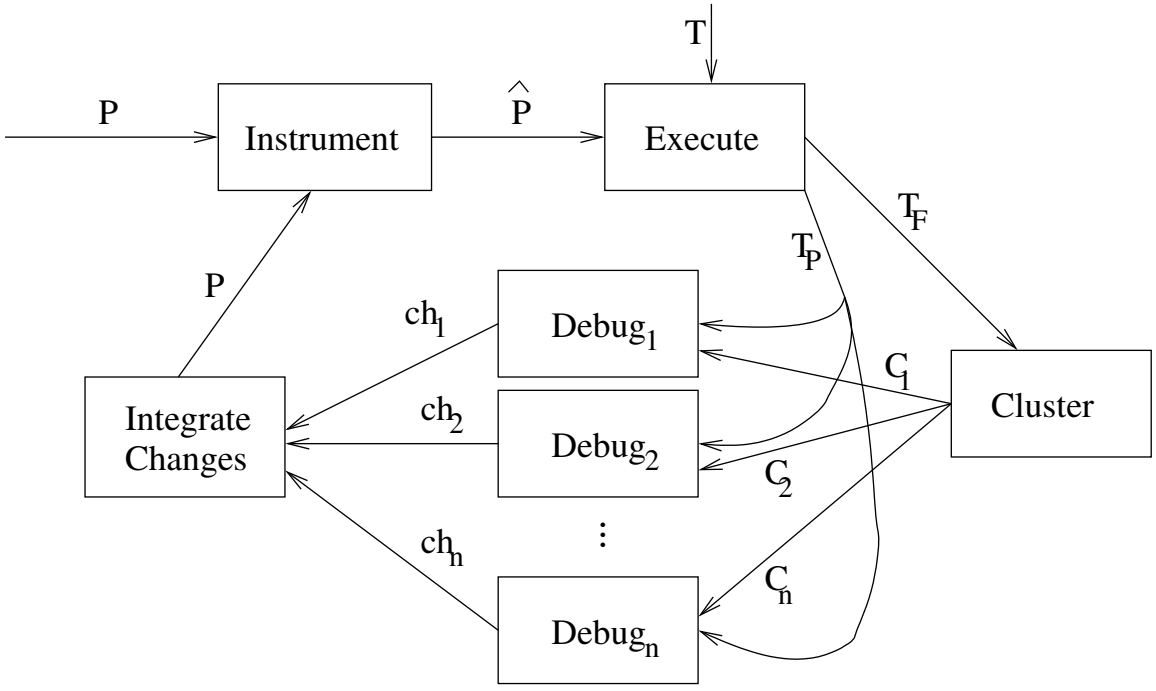

Figure 8: Technique for effective debugging in the presence of multiple faults.

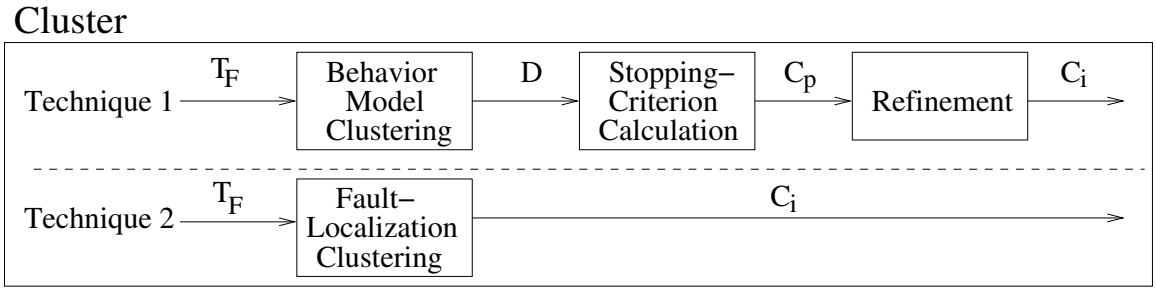

Figure 9: Two alternative techniques to cluster failed test cases.

C_i (described in Section 4.2.1.3). The first step is based on instrumentation profiles, and the second and third steps are based on fault-localization results.

4.2.1.1 Clustering profile-based behavior models

To group the failed test cases according to the likely faults that caused them, we use a technique for clustering executions based on agglomerative hierarchical clustering developed by Bowring and colleagues [12]. For each test case, this technique creates a behavior model that is a statistical summary of data collected during program execution. The specific models are Discrete-Time Markov Chains (DTMCs), and clustering occurs iteratively with the two most similar models merged at each iteration. Every execution is represented by its branching behavior. The branching profile of an execution is represented by the percentage of times that each branch of a predicate statement was taken. By using the percentage of times that each branch was taken, the branch profile is normalized—we call the result a *normalized branch profile*. Similarity is measured with a similarity metric—in this research, that metric is the sum of the absolute difference between matching transition entries in the two DTMCs being compared. Each pair of executions is assigned a similarity value that is the sum of the differences of the branch percentage profile. The technique initially sets the stopping criterion for the clustering to one cluster, so that the clustering proceeds until one cluster remains.

To illustrate clustering, consider Figure 10, which shows a dendrogram [24] that depicts an example clustering of ten execution models. A *dendrogram* is a tree diagram frequently used to illustrate the arrangement of clusters produced by a clustering algorithm. A dendrogram has a number of *levels*, each specifying a form of clustering. We denote level numbers by the number of clusters in that level. When proceeding from level n to level $n-1$, two clusters from level n are merged into one parent cluster in $n-1$. This clustering process produces one pair of branches between every

two dendrogram levels. The left side of the figure shows the ten individual failed test cases represented as $f1, ..., f10$. At each level of the dendrogram, the process of clustering the two most similar test cases is shown.[2] Initially, at level 10, failed test cases $f1, ..., f10$ are placed in clusters $c10.1, ..., c10.10$, respectively. Then, suppose the clustering algorithm finds that $c10.9$ and $c10.10$ have the most similar behavior models, and clusters them to get a new cluster, labeled as $c9.9$. This clustering results in nine clusters at level 9. Suppose further that, at level 8, the clustering algorithm finds that $c9.1$ and $c9.2$ are grouped to form $c8.1$. This clustering continues until there is one cluster, $c1.1$.

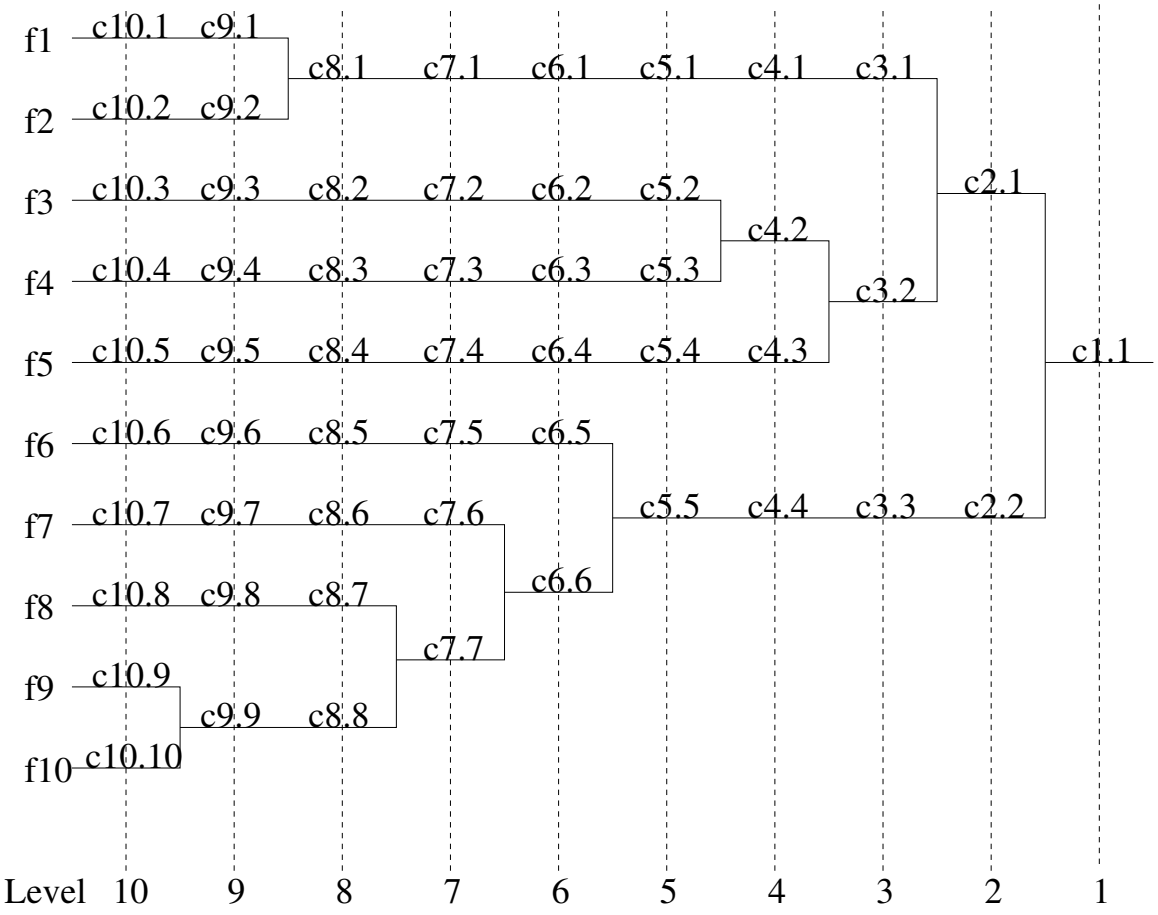

Figure 10: Dendrogram for 10 failed test cases.

[2]If multiple pairs are equally "most similar," one such pair is chosen at random.

The algorithm for our profile-based clustering technique is presented in Figure 11 with its supporting functions presented in Figures 12 and 13. In Figure 11, the algorithm ClusterProfile takes an initial set of clusters C as input. Each of these clusters $[C_1, C_2, ..., C_n]$ contain only one failed test case. In addition to the list of test cases that it contains, the cluster data structure has a representative, normalized execution profile. Line 2 stores the initial size of C in n. Lines 3-6 create the level of the dendrogram D that contains all singleton-cluster leaf nodes. Line 7 iterates for every level in the dendrogram being constructed, except the last level that will contain only one cluster. Line 8 initializes the variable m that records the least distance between any two clusters in this level. Lines 9-10 iterate over all possible cluster pairs. Line 11 checks whether the distance mapping δ_{ij} has been already computed for the two considered clusters C_i and C_j. If it has not been already computed, the value of δ_{ij} will be NIL and the distance will be computed and stored in δ on Line 12. Lines 14-18 compare δ_{ij} with the least known distance m for this level. If δ_{ij} is found to be less than m, m is redefined as δ_{ij} and the two clusters are recorded as M_i and M_j. Finally, Line 21 merges the two clusters M_i and M_j, updates the cluster list C, and augments the dendrogram D with another level including the new merged cluster.

The algorithm for computing the distance of two clusters is presented in Figure 12. The function ProfileDistance takes two clusters C_i and C_j as input. In addition to the list of test cases that each cluster contains, each contains a representative, normalized execution profile. Line 2 initializes the distance d between clusters C_i and C_j to zero. Line 3 iterates over all coverage entities E in the program. For each coverage entity, the absolute difference a of the profile values for that entity E_x of the two clusters C_i and C_j is computed on Line 4. Line 5 increases the distance d by a. After iterating over all entities, the final value of d specifies the distance between clusters C_i and C_j.

Algorithm: ClusterProfile

Input : C: a list of clusters $[C_1, C_2, ..., C_n]$ each containing one failed test case and each with its normalized execution profile

Output: D: a dendrogram

Declare: δ_{ij}: distance between cluster C_i and C_j, initialized to NIL for all possible pairs
M_i: cluster to be merged
M_j: cluster to be merged
m: least distance found between any two clusters for a level in the dendrogram

```
1  begin
2      n ← |C|
3      create a new level L_n in D
4      foreach cluster C_i in C do
5          add C_i to D as a leaf node at level L_n
6      end
7      for l ← n to 2 do
8          m ← ∞
9          foreach cluster C_i in C do
10             foreach cluster C_j in C where C_i ≠ C_j do
11                 if δ_ij = NIL then
12                     δ_ij ← ProfileDistance(C_i, C_j)
13                 end
14                 if δ_ij < m then
15                     m ← δ_ij
16                     M_i ← C_i
17                     M_j ← C_j
18                 end
19             end
20         end
21         Merge(M_i, M_j, C, D, l)
22     end
23 end
```

Figure 11: Profile-based clustering algorithm.

```
   Function: ProfileDistance
   Input  : C_i: a cluster of test cases along with a representative normalized
            execution profile
            C_j: a cluster of test cases along with a representative normalized
            execution profile
   Output : d: assigned distance between C_i and C_j
   Declare: ρ_ix: a profile value for entity E_x and cluster C_i
1  begin
2  |  d ← 0
3  |  foreach coverage entity E_x in the program do
4  |  |  a ← |ρ_ix − ρ_jx|
5  |  |  d ← d + a
6  |  end
7  end
```

Figure 12: `ProfileDistance` function used in the Profile-based clustering algorithm.

The algorithm for merging the two most-similar clusters is presented in Figure 13. The function Merge takes two clusters C_i and C_j as input. In addition to the list of test cases that each cluster contains, each cluster also contains a representative, normalized execution profile. Merge also takes the list of all clusters C as input, the current state of the dendrogram D, and the current level l in D. Lines 2-5 create a new cluster C_k and place the union of the test cases in each C_i and C_j in it. Lines 6-8 compute the representative, normalized execution profile for C_k. The profile ρ_{kx} for the coverage entity E_x and the new cluster C_k is computed as the average of the profiles ρ_{ix} and ρ_{jx}, which are the profiles for the coverage entity E_x for clusters C_i and C_j, respectively. Line 9 removes clusters C_i and C_j from the active set of clusters C. Line 10 creates a new level L_{l-1} in the dendrogram. Lines 11-14 places all remaining clusters in C into the new level L_{l-1} and connects its corresponding cluster in level L_l in D. Line 15 places the new cluster C_k in the active set of clusters C, and Line 16 places C_k in the new level L_{l-1} of D. Lines 17-18 connect C_k as the parent node in level L_{l-1} of each C_i and C_j in level L_l in D.

Function: MERGE

Input : C_i: a cluster of test cases along with a representative normalized execution profile
C_j: a cluster of test cases along with a representative normalized execution profile
C: a list of active clusters
D: an incomplete dendrogram
l: the current level number in the dendrogram

Output: C: the list of active clusters will be updated
D: the dendrogram will be augmented

Declare: C_k: the new cluster created by merging C_i and C_j
ρ_{ix}: a profile value for entity E_x and cluster C_i

```
1  begin
2      T_i ← test cases in cluster C_i
3      T_j ← test cases in cluster C_j
4      T_k ← T_i ∪ T_j
5      create cluster C_k containing test cases T_k
6      foreach coverage entity E_x in the program do
7          ρ_kx ← (ρ_ix + ρ_jx)/2
8      end
9      C ← C − {C_i, C_j}
10     create new level L_{l−1} in D
11     foreach cluster C_r in C do
12         include C_r in level L_{l−1} of D
13         create an edge in D from parent C_r in level L_{l−1} to child C_r in level L_l
14     end
15     C ← C ∪ {C_k}
16     include C_k in level L_{l−1} of D
17     create an edge in D from parent C_k in level L_{l−1} to child C_i in level L_l
18     create an edge in D from parent C_k in level L_{l−1} to child C_j in level L_l
19 end
```

Figure 13: `Merge` function used in the Profile-based clustering algorithm.

To demonstrate the clustering technique, Figure 14 shows control-flow graphs for one example program for three executions. The labels on the edges show the percent of times that each branch was taken by that one execution. For example, the first execution caused predicate P1's left branch to be taken 90% of the time and its right branch to be taken the other 10% of the time.

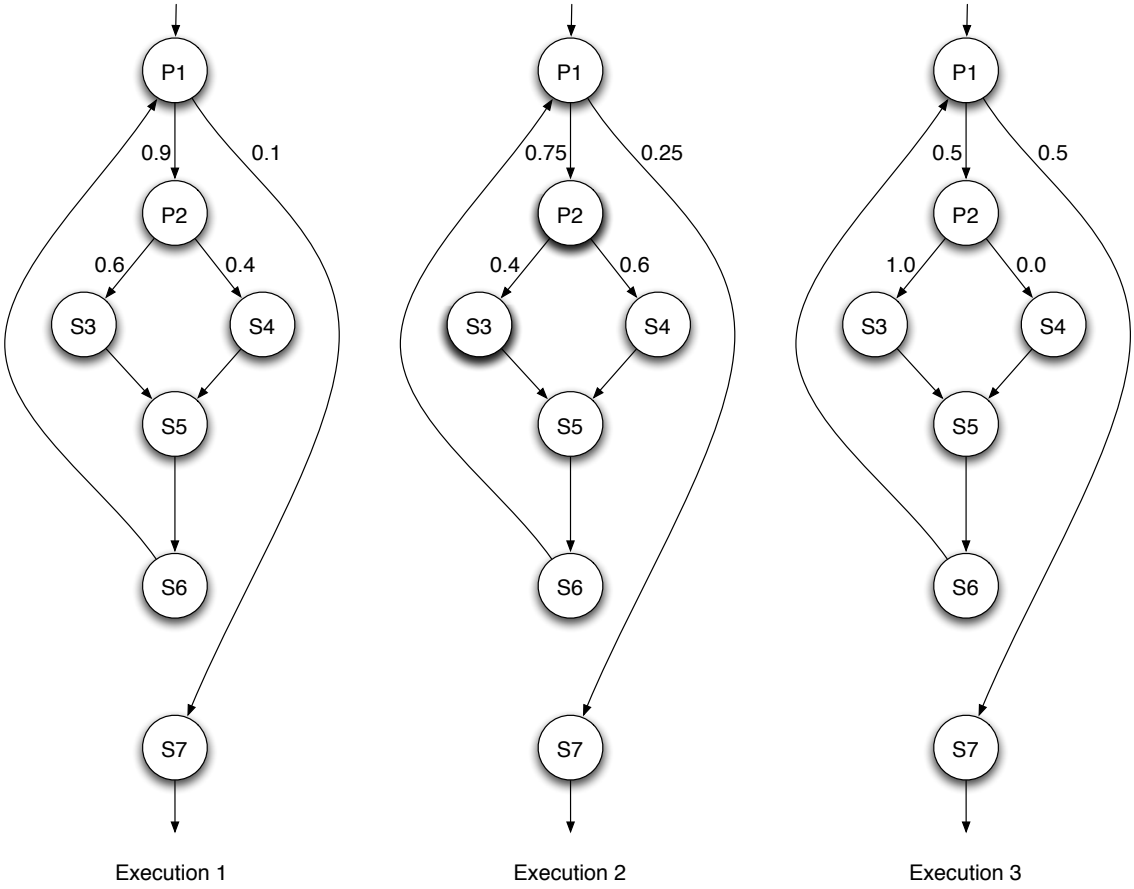

Figure 14: Three executions on one example program. The control-flow graph for the program is shown once for each execution. The branch labels show the percentage (as a probability) that each branch was taken during that execution

Table 1 shows the normalized branch profiles and their pair-wise differences. The columns marked *e1*, *e2*, and *e3* show the branch profiles taken from Figure 14. The next three columns show the pair-wise, absolute differences for each branch. For example, for *P1-l*, *e1* has a normalized branch profile of 0.9 and *e2* has a normalized

branch profile of 0.75. The absolute difference between these two values is $|0.9 - 0.75| = 0.15$. The last row shows the sum of all differences. For this example, executions *e1* and *e2* are most similar because their total difference was the least of these three.

Table 1: Example branch profiles from Figure 14 and their pair-wise differences.

	$e1$	$e2$	$e3$	$\|e1 - e2\|$	$\|e2 - e3\|$	$\|e1 - e3\|$
P1-l	0.9	0.75	0.5	0.15	0.25	0.4
P1-r	0.1	0.25	0.5	0.15	0.25	0.4
P2-l	0.6	0.4	1.0	0.2	0.6	0.4
P2-r	0.4	0.6	0.0	0.2	0.6	0.4
Total difference				0.7	1.7	1.6

Because executions *e1* and *e2* were found to be most similar, they are clustered together. A new model represents the new cluster. The clustered model for the cluster containing *e1* and *e2* is represented by the mean of the normalized branch profiles of its two constituent members. Thus, for this example, for the branches {*P1-l*, *P1-r*, *P2-l*, *P2-r*}, the clustered model would be $\{0.825, 0.175, 0.5, 0.5\}$. That cluster would next be compared against the singleton cluster containing *e3*. Because there is only one pair-wise difference, it would be found to be the minimum one and these two models would be clustered together creating one large composite cluster containing all *e1*, *e2*, and *e3*.

Conventionally, a good stopping criterion for the clustering, which is difficult to determine [24], is based on the practitioners' domain knowledge. Because our domain is debugging, we have developed a technique that inputs the dendrogram and computes the stopping criterion based on fault-localization information. We describe this stopping criterion in the next section.

4.2.1.2 Using fault localization to stop clustering

We use a fault-localization-based algorithm for this secondary assessment of the clustering. This implementation uses the TARANTULA technique to provide a prediction of the location of the fault for each specialized test suite. A number of other fault-localization techniques might also be used for this purpose (e.g., [48, 49]).

Figure 10 shows the process of grouping clusters until one cluster remains. Between these two extremes is where the richness in the representation lies. Unless there is only one behavior represented by the test cases, at some point during this clustering two clusters are merged that are not similar. In the context of fault localization, unless there is only one fault, at some point in the clustering process the failed test cases that fail due to one fault are merged with failed test cases that fail due to another fault. The goal of the technique is to stop the clustering process just before this type of clustering occurs.

The technique identifies the clustering-stopping criterion by leveraging the fault-localization results. The technique computes the fault-localization ranks (the ranking of all statements in the program from most suspicious to least suspicious based on the Tarantula heuristic) for each individual failed test case f (shown at the left side of Figure 10) using a test suite of all passed test cases T_P with that one failed test case, $\{f\} \cup T_P$. Then, every time a new cluster C_i is created by merging two clusters, the technique calculates the fault-localization ranks using the members of that cluster and the passed test cases, (i.e. $C_i \cup T_P$). Thus, with regard to a dendrogram, such as Figure 10, the technique computes fault-localization rankings at every merge point of two clusters.

Using these fault-localization rankings at all merge points in the dendrogram, the technique uses a similarity measure to identify when the clustering process appears to lose the ability to find a fault—that is, when it clusters two items that contribute to find a different suspicious region of the program. To measure the similarity of two

fault-localization results, we first define the suspicious area of the program as the set of statements of the program that are deemed "most suspicious" for each of the results. This process is depicted in Figure 15.

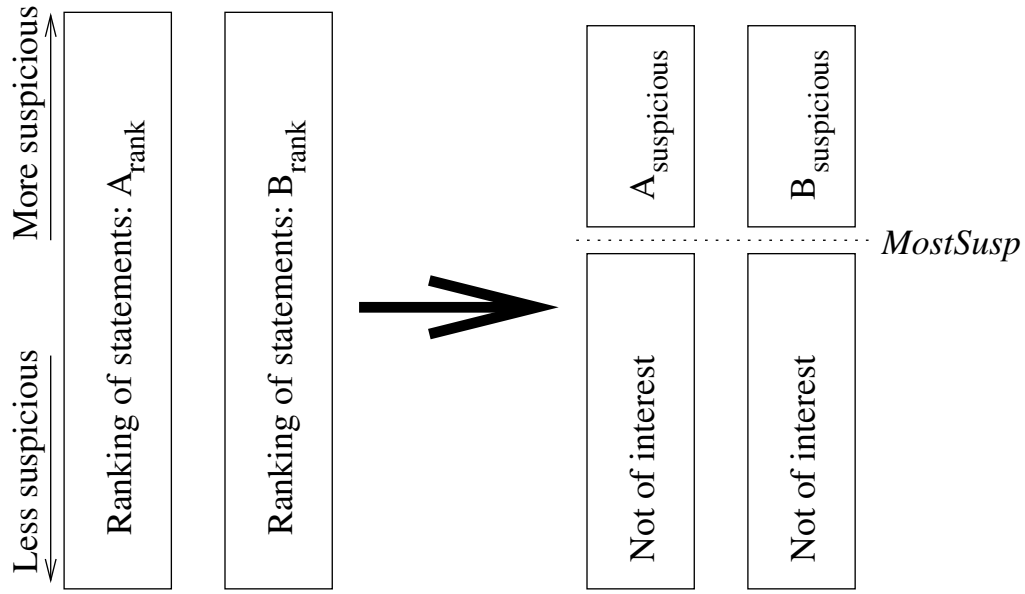

Figure 15: Similarity of fault-localization results is performed by identifying two sets of interest $A_{suspicious}$ and $B_{suspicious}$ and performing a set similarity.

To decide whether two fault-localization results identify the same suspicious region of the program, we must establish the threshold that differentiates the *most suspicious* statements from the statements that are *not of interest*. We call this threshold *MostSusp*. For example, we may assign the value of 20% to *MostSusp*—this means that the top 20% of the suspicious statements in the ranking are in the most suspiciousness set, and that the lower 80% are not of interest.

To compare the two sets of statements, we use a set-similarity metric called *Jaccard set similarity*.[3] The Jaccard metric computes a real value between 0 (completely dissimilar) and 1 (completely similar) by evaluating the ratio of the cardinality of the

[3]We experimented with other similarity metrics including "Ulam's distance" which considers the order of the list, but found that the Jaccard metric performed as well as, and often better, than the others, while being more efficient to compute.

intersection of these sets to the cardinality of the union of these sets. The similarity of two sets, A and B, is computed by the following equation:

$$similarity(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{4.2.1}$$

To determine whether the two sets are *similar* or *dissimilar*, we establish the threshold for the similarity metric. We call this threshold Sim. For example, we may assign the value of 0.7 to Sim—this means that two sets of suspicious statements will be in the same cluster if their similarity value is at or above 0.7. We envision that in practice these thresholds, $MostSusp$ and Sim, would be determined during a training phase that shadows the debugging process.

To determine where to stop the clustering, the technique traverses the dendrogram in reverse—starting at the final cluster. At each step, the technique examines the merged clusters at that level, and computes the similarity, using Equation 4.2.1, of the fault-localization rankings of the merged cluster with its constituent clusters. When at least one of the constituent clusters is dissimilar to the merged cluster, the traversal has found new information, and thus, the traversal continues (i.e., this is not the stopping point for the clustering). For example, in Figure 10, the fault-localization result of $c1.1$ is compared with the fault-localization result of each $c2.1$ and $c2.2$ using Equation 4.2.1. If $c1.1$ is dissimilar to either $c2.1$ or $c2.2$, the traversal continues.

Our experience with our empirical evaluation (presented in Chapter 7) has led to the observation that the typical result of this analysis is that the composite clusters are often dissimilar to at least one of their constituents on the more clustered end of the dendrogram (the right side of Figure 10), and that the composite clusters are rarely dissimilar to their constituents on the less clustered end of the dendrogram (the left side of Figure 10). At some point in between, the clustering begins to show the constituent clusters beginning to differ from the composite clusters in terms of their fault-localization results. This is the point at which we stop the clustering.

The algorithm to determine the stopping point for the profile-based clustering is

presented in Figure 16. The algorithm DETERMINESTOPPINGPOINT takes a dendrogram D as input. It also takes the two thresholds $MostSusp$ and Sim as input. Lines 2-5 compute the Tarantula metrics, *suspiciousness* and *confidence*, for each coverage entity in E and for each cluster in C. To do this, the algorithm calls the ASSIGNMETRICS algorithm defined in Figure 2. Line 6 initializes a variable n to the number of levels in the dendrogram D. Line 7 iterates over the levels in D, starting from the most-clustered level (level 1) to the second-to-least-clustered level (level $n-1$). To illustrate this concept of "most clustered" and "least-clustered", consider again Figure 10. The most-clustered level is level 1 on the right side of the figure. The least-clustered level is level 10 on the left side of the figure. In the algorithm, Lines 8-10 assign p as the newly merged parent cluster for level l and c_1 and c_2 as the two constituent, child clusters in level $(l+1)$ for p. Lines 11-12 compute the similarity of p with c_1 and the similarity of p with c_2. Lines 13-15 determine if p is similar to both children, and if so, return the current level l as the stopping point.

To determine the similarity of the parent cluster p and the child clusters c_1 and c_2, the function FAULTSIMILARITY is used. The algorithm for this function is presented Figure 16. FAULTSIMILARITY takes two clusters C_i and C_j as input as well as the $MostSusp$ threshold. The clusters C_i and C_j contain the sorted list of coverage entities, from most suspicious to least suspicious according to the ASSIGNMETRICS algorithm (defined in Figure 2). Lines 2-3 create two sets s_i and s_j which contain the top $MostSusp$ coverage entities from the sorted list of entities from each C_i and C_j, respectively. Line 4 computes the similarity of these two sets using the Jaccard similarity metric.

Algorithm: DETERMINESTOPPINGPOINT

Input : D: a dendrogram describing the clustering of test cases, along with each cluster's associated profile
$MostSusp$: percentage threshold distinguishing the most suspicious entities from the entities not of interest
Sim: percentage threshold for the similarity metric

Output: l: the level in D for stopping the clustering

Declare: $M_C[E, T_C]$: coverage matrix for cluster C, the program's coverage entities E, and C's test cases T_C
P: a list of boolean values $[P_1, P_2, ..., P_m]$ specifying whether each test case in T passed

```
1  begin
2      foreach cluster C in D do
3          AssignMetrics(M_C, P)
4          sort coverage entities E from most suspicious to least
5      end
6      n ← number of levels in D
7      for l ← 1 to (n − 1) do
8          p ← merged parent cluster for level l in D
9          c_1 ← child cluster 1 of p in level (l + 1) in D
10         c_2 ← child cluster 2 of p in level (l + 1) in D
11         δ_pc1 ← FaultSimilarity(p, c_1, MostSusp)
12         δ_pc2 ← FaultSimilarity(p, c_2, MostSusp)
13         if δ_pc1 > Sim and δ_pc2 > Sim then
14             return l
15         end
16     end
17 end
```

Function: FAULTSIMILARITY

Input : C_i: cluster with its ranked list of entities
C_j: cluster with its ranked list of entities
$MostSusp$: percentage threshold distinguishing the most suspicious entities from the entities not of interest

Output: d: assigned distance between C_i and C_j

```
1  begin
2      s_i ← top MostSusp most suspicious entities for C_i
3      s_j ← top MostSusp most suspicious entities for C_j
4      d ← (s_i ∩ s_j) / (s_i ∪ s_j)
5  end
```

Figure 16: Clustering stopping-point algorithm.

4.2.1.3 Using fault-localization clustering to refine clusters

After the clusters are identified using profiles and fault-localization results, the technique performs one additional refinement. Occasionally, similar fault-localization results are obtained on multiple branches of a dendrogram. To merge these similar clusters, the technique groups clusters that produce similar fault-localization results.

Consider, for example, Figure 17, the same dendrogram as Figure 10 except that each branch is labeled with the fault to which it is focused.[4] For example, at clustering Level 10, six of the clusters produce fault-localization results that are focused at Fault 2, two produce fault-localization results that are focused at Fault 1, and one produces fault-localization results that are focused at Faults 3 and 4. On the other side of the dendrogram, at Level 1, the one cluster produces fault-localization results that are focused at Fault 1.

In this dendrogram, clustering Level 5 gives the maximum clustering without diminishing the clusters' abilities to locate all faults. Note that for each of Faults 1, 3, and 4, there is one cluster that produces fault-localization results that target that fault. However, for Fault 2, there are two such clusters. We want to merge these two clusters to produce one cluster that targets Fault 2.

To identify the places where this refinement of the clustering can be applied, the technique performs a pair-wise comparison of the fault-localization results of the clusters at the stopping-point level of the dendrogram. For this comparison, we use the *Jaccard similarity* parameterized for this task. We then merge the similar clusters.

For example, in Figure 10, consider that the stopping point of the clustering was determined to be best at Level 5. A pair-wise similarity would be calculated for the five clusters at this level by inspecting the similarity of the suspicious statements that each targets. If it found that clusters $c5.4$ and $c5.5$ were similar, these would be

[4]Of course, this is hidden knowledge that the technique cannot know. However, I use it for discussion of the goal of the refinement.

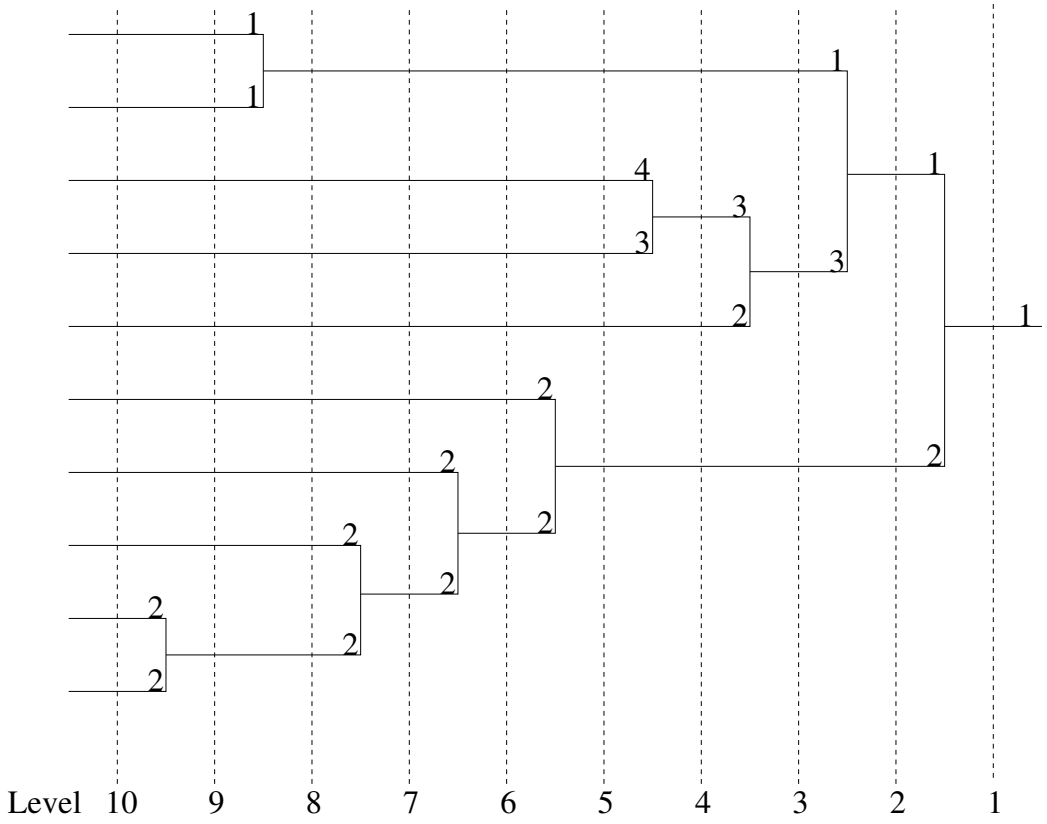

Figure 17: Dendrogram with fault number of the best exposed fault.

combined to produce the final set of fault-focusing clusters.

4.2.2 Complexity Analysis of Profile-based Clustering

The profile-based clustering performs a pair-wise comparison of the clusters. At the beginning of the clustering, there is one singleton cluster for every failed test case. There will be $\binom{t}{2}$ comparisons, which is equivalent to $t(t-1)/2$ comparisons. At the second level in the dendrogram, there are $t-1$ clusters ($t-2$ singleton clusters and one new merged cluster containing two test cases). The new cluster must be compared against each other cluster, but the others do not need to be re-compared to each other because those results can be stored and reused from the previous level in the dendrogram. Thus, at this second level, there are an additional $t-2$ comparisons. The number of comparisons for Level 2 through Level t (there are always exactly t levels in the dendrogram) is $(t-2)+(t-3)+(t-4)+...+1$ which is equal to $(t-1)(t-2)/2$. For all levels of the dendrogram, the number of comparisons is $t(t-1)/2+(t-1)(t-2)/2 = t^2-t+1$. Each comparison requires that each branch's profile be differenced between the two compared clusters. Each branch's difference is a constant-time operation (specifically, a subtraction operation), with a cost C. Suppose there are s branches in the code (a measure of the size of the program). Each cluster comparison will require Cs time. Thus, the total time to perform all comparisons and build the dendrogram is $Cs(t^2-t+1)$. The run-time complexity is $O(t^2 s)$.

To determine the stopping point of the clustering, the rankings from the fault-localization technique must be computed for possibly every cluster represented in the dendrogram. The dendrogram contains exactly t singleton clusters and $t-1$ merged clusters. For the $2t-1$ clusters, the fault-localization technique must be performed on the fault-focusing cluster that that cluster represents. If using Tarantula and the time and complexity analysis from Section 3.5.1, each cluster's fault-localization calculation

requires $s(t+C)$ time and complexity of $O(tn)$, where s is the number of coverage entities, t is the number of test cases, and C is a constant. The total time to generate all the fault-localization results is $(2t-1)*s(t+C)$ which is $O(t^2s)$ complexity. Each merge point is compared against both of the clusters that contributed to it. Because there are $t-1$ merged clusters, there are $2(t-1)$ such comparisons. Each comparison requires (1) the sorting of the coverage entities which in the worst case is $O(s \log s)$, and (2) the set similarity calculation (using the Jaccard measure which is computed with set intersection and set union operations) which is computed in $O(s)$. The complexity for the comparisons is $O(t * [(s \log s) + s]) = O(ts \log s)$. The complexity of the determination of the stopping point is composed of the fault-localization result computation and the comparisons. Thus, the total complexity for determining the stopping point of the clustering is $O(t^2s + ts \log s)$. The overall run-time complexity for building the dendrogram and determining the stopping point is $O(t^2s + ts \log s)$.

4.2.3 Clustering Based on Fault-localization Results

The second fault-focusing technique, shown as Technique 2 in Figure 9, uses only the fault-localization results. The technique first computes the fault-localization suspiciousness rankings for the individual failed test cases, T_F, and uses the *Jaccard similarity* metric to compute the pair-wise similarities among these rankings. Then, the technique clusters by taking a closure of the pairs that are marked as similar.

The algorithm to calculate the fault-localization-based clustering is presented in Figure 18. The algorithm, ClusterFault, takes an initial set of clusters C as input, as well as the $MostSusp$ and Sim thresholds. Lines 2-5 initialize a graph G with each cluster from C as a node, and compute for each cluster the ranking of entities from most-suspicious to least-suspicious according to the AssignMetrics algorithm (defined in Figure 2). Lines 7-14 iterate over all cluster pairs. For each cluster pair, Line 10 computes a similarity measure using the FaultSimilarity function (defined

in Figure 16). Lines 11-13 create an edge in graph G between the two clusters in the cluster pair. Lines 16-22 compute the final set of clusters F by calculating the nodes that are reachable from each other. All clusters that are reachable are grouped into a cluster. Lines 16-22 iterates until the initial set of clusters C is empty. Line 17 takes any cluster C_i from C. Line 18 computes the set S of clusters that are reachable from C_i. Line 19 puts cluster C_i in S; the set S now forms one of the final clusters. Line 20 adds S to the final set of clusters F. Line 21 removes all clusters in S from C.

For example, consider Figure 19, which shows the same ten failed test cases depicted in Figure 10. Each failed test case is depicted as a node in the figure. The technique combines each failed test case with the passed test cases to produce a test suite. The technique uses Tarantula to produce a ranking of suspiciousness for each test suite, and these rankings are compared using the Jaccard metric in the same way described in Sections 4.2.1.2 and 4.2.1.3. The technique records the pairs of clusters that are identified as similar (above the similarity threshold). In Figure 19, a pair-wise similarity between failed test cases is depicted as an edge. The technique produces clusters of failed test cases by taking a closure of the failed test cases that were marked as similar. Thus, any test case that is reachable from another test case will be in the same cluster. Using the example in Figure 19, test case nodes that are reachable over the similarity edges are clustered together. In this example, failed test cases $f1$ and $f2$ are combined to a cluster, $f6$, $f7$, $f8$, $f9$, and $f10$ are combined to a cluster, and $f3$, $f4$, and $f5$ are each singleton clusters.

4.2.4 Complexity Analysis of the Fault-Localization-based Clustering

The fault-localization clustering technique performs a pair-wise comparison of each pair of clusters. At the beginning of the clustering, there are t (where t is the number of failed test cases) fault-localization results. Each of these fault-localization results was computed by calculating the suspiciousness and confidence of each coverage entity.

Algorithm: ClusterFault

Input : C: a list of initial clusters $[C_1, C_2, ..., C_n]$
$MostSusp$: percentage threshold distinguishing the most suspicious entities from the entities not of interest
Sim: percentage threshold for the similarity metric

Output: F: the set of final clusters

Declare: $M_C[E, T_C]$: coverage matrix for cluster C, the program's coverage entities E, and C's test cases T_C
P: a list of boolean values $[P_1, P_2, ..., P_m]$ specifying whether each test case in T passed
G: an empty graph of cluster nodes

```
1  begin
2      foreach cluster C_i in C do
3          create a node for C_i in G
4          AssignMetrics(M_C, P)
5          sort coverage entities E from most suspicious to least
6      end
7      n ← |C|
8      for i ← 1 to (n − 1) do
9          for j ← i to n do
10             δ ← FaultSimilarity(C_i, C_j, MostSusp)
11             if δ > Sim then
12                 create an undirected edge in G from C_i to C_j
13             end
14         end
15     end
16     while C ≠ ∅ do
17         C_i ← a cluster from C
18         S ← set of all clusters reachable from C_i in G
19         S ← S ∪ {C_i}
20         F ← F ∪ {S}
21         C ← C − S
22     end
23 end
```

Figure 18: Fault-localization-based clustering algorithm.

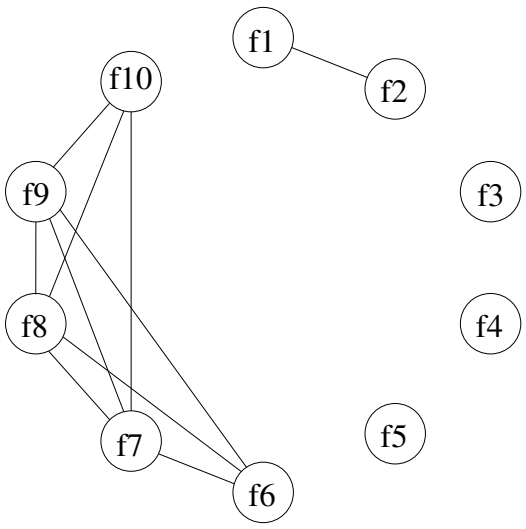

Figure 19: Graph where each node represents a failed test case and edges represent pairs that are deemed similar. Clusters are formed by all nodes that are mutually reachable.

The amount of effort to generate each of these fault localization results is linear in the size of the program in terms of the number of coverage entities and linear in the size of the test suite. Let s represent the number of coverage entities in the program. As shown in Section 3.5.1, the Tarantula algorithm's complexity is $O(ts)$. Because the algorithm has to calculate t such results, the complexity of this first stage is $O(t^2s)$. After this, the algorithm sorts the coverage entities from most suspicious to least suspicious. To sort over all fault-localization results, this step would be $O(ts \log s)$. Finally, the algorithm performs a pair-wise comparison of these results. Just like the first level of the profile-based clustering, the algorithm performs $O(t^2)$ such comparisons. Each comparison will require $O(s)$ steps to perform the Jaccard distancing. The overall run-time complexity is $O(ts+ts \log s+st^2) = O(t^2s+ts \log s)$.

4.3 Parallel Approach to Debugging

A developer can inspect a single failed test case to find its cause using an existing debugging technique (e.g., [17, 78]), or she can utilize all failed test cases using a

fault-localization technique (e.g., [42, 43, 48, 49]). Regardless of the technique chosen, after a fault is found and fixed, the program must be retested to determine whether previously failed test cases now pass. If failures remain, the debugging process is repeated. We define this one-fault-at-a-time mode of debugging and retesting *sequential debugging*.

In practice, however, there will typically be more than one developer available to debug a program, particularly under urgent circumstances such as an imminent release date. Because, in general, there may be multiple faults whenever a program fails on a test suite, an effective way to handle this situation is to create parallel work flows so that multiple developers can each work to isolate different faults, and thus, reduce the overall time to a failure-free program. Like the parallelization of other work flows, the principal problem of providing parallel work flows in debugging is determining the partitioning and assigning of subtasks. The partitioning requires an automated technique that can detect the presence of multiple faults and map them to sets of failed test cases (i.e., clusters) that can be assigned to different developers.

To parallelize the debugging effort, we devised a technique that we call *parallel debugging* that is an alternative to sequential debugging. Parallel debugging automatically partitions the set of failed test cases into clusters that target different faults, called *fault-focusing clusters*, using behavior models and fault-localization information created from execution data. Each fault-focusing cluster is then combined with the passed test cases to get a *specialized test suite* that targets a single fault. Consequently, specialized test suites based on fault-focusing clusters can be assigned to developers who can then debug multiple faults in parallel. The resulting specialized test suites might provide a prediction of the number of current, active faults in the program.

The main benefit of this technique for parallel debugging is that it can result in decreased time to a failure-free program; the empirical evaluation (in Chapter 7)

supports this claim for the subject program used in the evaluation. When resources are available to permit multiple developers to debug simultaneously, which is often the case, specialized test suites based on fault-focusing clusters can substantially reduce the time to a failure-free program while also reducing the number of testing iterations and their related expenses. Another benefit is that the fault-localization effort within each cluster is more efficient than without clustering. Thus, the debugging effort yields improved utilization of developer time, even if performed by a single developer. Our empirical evaluation shows that, for the subject used, using the clusters provides savings in effort, even if debugging is done sequentially. A third benefit is that the number of clusters is an early estimate of the number of existing active faults.

A final benefit is that the technique automates a debugging process that naturally occurs in current practice. For example, on bug-tracking systems for open-source projects, multiple developers are assigned to different faults, each working with a set of inputs that cause different known failures. The technique improves on this practice in a number of ways. First, the current practice requires a set of coordinating developers who triage failures to determine which appear to exhibit the same type of behavior. Often, this process involves the actual localization of the fault to determine the reason that a failure occurred, and thus a considerable amount of manual effort is needed. The techniques can categorize failures automatically, without the intervention of the developers. This automation can save time and reduce the necessary labor involved. Second, in the current practice, developers categorize failures based on the failure output. The techniques look instead at the execution behavior of the failures, such as how control flowed through the program, which may provide more detailed and rich information about the executions. Third, the current practice involves developers finding faults that cause failures using tedious, manual processes, such as using print statements and symbolic debuggers on a single failed execution. Our techniques can automatically examine a set of failures and suggest likely fault locations in the

program.

4.4 Sequential and Parallel Debugging

The sequential and parallel debugging modes, described in Section 4.3, are analogous to many types of sequential and parallel work flows. One such example is the parallelization of computation on multi-processor computers. On a multi-processor computer, a task is divided into subtasks that are processed simultaneously with coordination between the processors. There is a cost of this coordination, and thus, the total processing effort is often higher in the parallel computation than the sequential one. However, because of better utilization of the processors and the divide-and-conquer strategy, the task can often complete faster when computed in parallel.

To illustrate, consider Figures 20 and 21, which represent sequential and parallel computation of a task, respectively. In the figures, the solid arrows represent the cost of the subtasks, and the dotted arrow in Figure 21 represents the overhead of performing the tasks in parallel. The figures illustrate that, whereas there is some cost associated with the parallelization of the task, with parallel processing, the overall time to complete the task can be much less than in the sequential processing of the task. Also, Figure 20 shows that in the sequential processing, only one of the processors is utilized in the computation of this task.

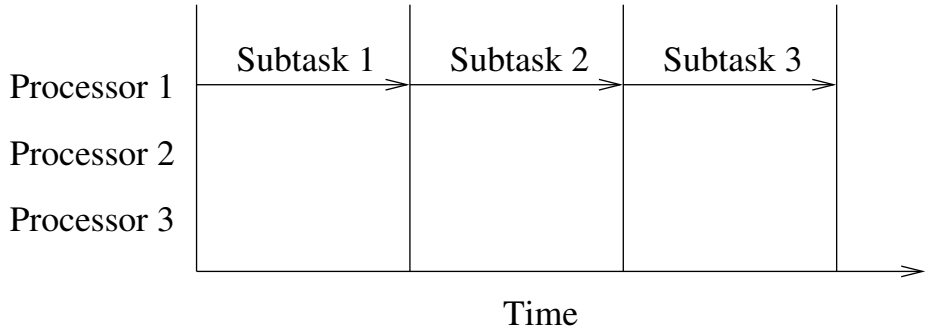

Figure 20: Sequential processing of a task.

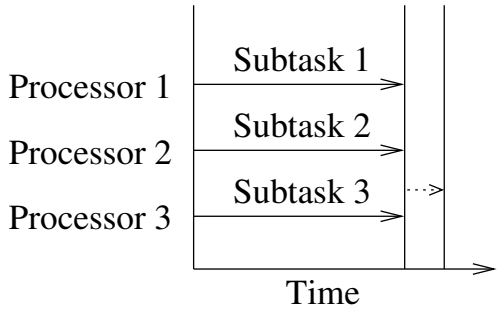

Figure 21: Parallel processing of a task.

Figures 20 and 21 illustrate the two dimensions of this parallelization—the completion time of the task and the degree of parallelization that was accomplished for the task. The "width" of these figures depicts the time dimension, and the "height" depicts the parallelization dimension. In Figure 20, the width shows that the task took a relatively long time to complete, and the height shows that there was little parallelization of the task—in this case, there was no parallelization of the task. In Figure 21, the width shows that the task took a relatively short time to complete, and the height shows that the task was parallelized to a large degree—in this case, the task was fully parallelized. For the parallelization of the debugging task, we can also measure these two dimensions. The completion of the debugging task can be measured as the time to debug the faults causing the failures, and the degree of parallelization of the debugging task can be measured as the number of developers that can simultaneously debug the program.

Like the parallelization of a computation task, some debugging subtasks, such as locating one fault, can dominate other tasks. For example, a program that contains four faults may cause a number of test cases to fail. Upon inspection, we may find that all of the failed test cases fail due to one fault. After that dominating fault is found and fixed, the program is re-tested. This re-testing reveals that there are still a number of failed test cases, but these failed test cases are now caused by the remaining three faults. This phenomenon is illustrated in Figure 22. In the example,

Fault 1 must be located and fixed before Faults 2, 3, and 4 can be located and fixed because all failed test cases fail due to Fault 1. Only after Fault 1 is fixed do Faults 2, 3, and 4 manifest themselves as failures.

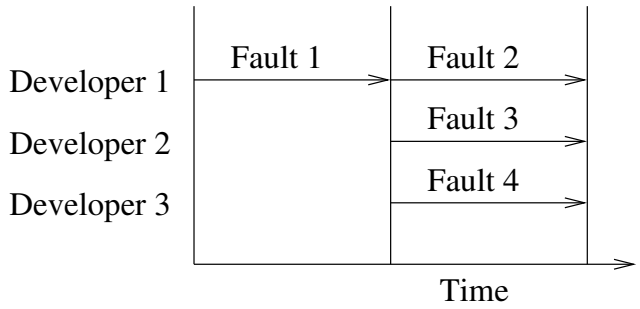

Figure 22: Fault 1 dominates Faults 2, 3, and 4.

Unlike the parallelization of a computation task, the cost for each fault subtask can change as a result of the parallelization. In fact, we found empirically that the fault subtasks are often more efficient in the parallelized version. In the parallelized version, the test suite for each fault subtask is generated specifically for that fault. Thus the fault localization is often more effective at locating that fault than the non-specialized, full test suite.

Consider the example presented in Figure 5 on page 40. In the traditional, sequential mode of debugging, the developer would be aware that there were four failed test cases, but would be unaware of the number of faults that caused them. Thus, a typical, sequential process that she follows might be:

1. Examine the statement at the highest level of suspicion: statement 10. She would realize that statement 10 was, in fact, faulty and would correct the bug.

2. Rerun the test suite to determine whether all of the faults were corrected. She would witness that two of the failed test cases now pass and two of the formerly failed test cases still fail. Figure 23 depicts the coverage and new, recomputed fault-localization results.

	Test Cases												
	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	suspiciousness	confidence	
mid() { int x,y,z,m;	3,3,5	1,2,3	3,2,2	5,5,5	1,1,4	5,3,4	3,2,1	5,4,2	2,1,3	5,2,6			rank
1: read("Enter 3 numbers:",x,y,z);	●	●	●	●	●	●	●	●	●	●	0.50	1.0	7
2: m = z;	●	●	●	●	●	●	●	●	●	●	0.50	1.0	7
3: if (y<z)	●	●	●	●	●	●	●	●	●	●	0.50	1.0	7
4: if (x<y)	●	●			●	●			●	●	0.67	1.0	3
5: m = y;		●									0.00	0.13	12
6: else if (x<z)	●				●	●			●	●	0.73	1.0	2
7: m = y; // fault1. correct: m=x	●				●				●	●	0.80	1.0	1
8: else			●	●			●	●			0.00	0.5	9
9: if (x>y)			●	●			●	●			0.00	0.5	9
10: m = y; // fault2 corrected			●				●	●			0.00	0.38	10
11: else if (x>z)				●							0.00	0.13	12
12: m = x;											0.00	0.0	13
13: print("Middle number is:",m);	●	●	●	●	●	●	●	●	●	●	0.50	1.0	7
} Pass/Fail Status	P	P	P	P	P	P	P	P	F	F			

Figure 23: Example *mid()* and all test cases after *fault2* was located and fixed.

3. Examine the statement at the highest level of suspicion: statement 7. She would realize that it was, in fact, faulty and would correct the bug.

4. Rerun the test suite. In this case, she would witness that all test cases pass.

Consider again the example in Figure 5. To demonstrate the utility of parallel debugging, assume that there exists a technique that can automatically determine that there are two distinct types of failures in this program and can automatically cluster them. The groupings of "Cluster 1" and "Cluster 2" are depicted in Figure 5. Given this clustering, a test suite can be generated for each cluster by combining all passed test cases with each cluster, and the fault-localization results can be calculated on this new, specialized test suite. The specialized test suites are shown in Figures 6 and 7. Each of these test suites and fault-localization results can be given to a different developer to debug. A parallel process that they follow in this circumstance might be:

1. Examine the statements at the highest level of suspicion: statement 7 for one developer and statement 10 for the other developer. They would each realize that those were, in fact, faulty and would correct them.

2. Rerun the test suite to determine if all of the faults were corrected. In this case, they would witness that all of the test cases pass.

This example demonstrates how an automated technique may reduce the overall time to achieve a failure-free program. Also notice that $fault1$ on Statement 7 was made more noticeable by the removal of the "noise" generated by $fault2$ on Statement 10 without the need to actually correct $fault2$.

4.5 Related Work

There are two broad areas of related work: the clustering of executions and the determination of which developer would be responsible for each fault-localization result. This section describes the related work in each of these areas.

4.5.1 Clustering Executions

The main component of the technique is the automatic clustering of failed executions according to their causes. Dickinson and colleagues show that clustering of executions can isolate failed executions from passed executions [22]. In later work, Podgurski and colleagues show that profiles of failed executions can be automatically clustered according to similar causes or faults [57]. Their approach depends on a supervised classification strategy informed by multivariate visualizations that assist the practitioner. In contrast, our technique is completely automated and attempts to cluster failed executions according to their root cause by combining information from execution profiles with information about the relative failure-causing suspiciousness of lines of code.

Bowring and colleagues present a technique to cluster executions using discrete-time Markov chains [12]. The goal of this work is to automatically classify future executions based on training information using active learning. The technique and implementation by Bowring and colleagues were extended and described in Section 4.2.1 and used for the experimentation described in Chapter 7.

Zheng and colleagues present an approach to finding bug predictors in the presence of multiple faults [79]. The authors show that test runs can be clustered to give a different bug-predictor profile or histogram. They also present a result that is similar to my findings: that some bug predictors dominate others—they call these predictors *super bug predictors*. We found similar results, although from a different perspective: we found that some faults prevent others from being active. Beyond this, our work differs from theirs in that our work presents a methodology for debugging multiple faults in parallel. Also, our work presents an experiment and metrics (in Chapter 7) describing the costs of debugging multiple faults.

Liu and Han present two pair-wise distance measures for failed test cases [50]. They demonstrate the difference between a profile-based distance measure (usage mode) and fault-localization-based distance measure (failure mode) by means of multidimensional-scaling plots. For the subject programs and plots that they present, they propose that the fault-localization-based distance measure is better able to isolate failures caused by different faults. Our work differs from theirs in a number of ways. First, unlike their multidimensional plots of executions, our work provides an automatic way to cluster failed test cases without interpretation by the developer. Second, our experiments do not confirm their finding that profile-based distances are inferior to fault-localization-based distances. Although, we cannot generalize to other programs, our experiments show that each type of clustering may have its own strengths. Finally, their work targets a sequential-mode of debugging by removing faults that are creating noise making it difficult to find the most dominant fault at

each iteration. Our work aims to enable the parallelization of the debugging task.

On the topic of clustering executions, note that the clustering of failures is related to a standard practice used by many developers. This failure clustering can be seen by considering one of many online bug-tracking systems. In such a bug-tracking system, failure reports are often assigned to different developers. Additionally, some failure reports are marked as *duplicates* of other failure reports. These duplicates, along with the failure report that they duplicate, are simply a manual form of clustering. This manual form of clustering requires that a coordinating developer must identify some characteristics that uniquely identify failures caused by different faults. In fact, often the coordinating developer must actually locate the fault that is causing the failures before she can group and then assign them. Our fully automatic technique can help with this manual approach by suggesting (1) likely fault locations, (2) when failure reports are duplicates, and (3) when marked duplicates may be incorrectly marked this way.

4.5.2 Determining Responsibility

On the topic of determining responsibility for fixing faults, Ren and colleagues [59] present the Chianti system that suggests which change to the program likely caused the failures. Anvik and colleagues [5] use source-code change management logs to suggest which developer might be most appropriate to address a particular failure. This area of research for determining responsibility for fixing faults is orthogonal to our goals for debugging in parallel. However, we believe it is complimentary; such techniques might work well along with the partitioning of the failures.

CHAPTER V

VISUALIZATION

Comprehension of fault-localization results may be difficult due to the amount of data that is produced by a technique like Tarantula. This chapter presents a visualization that is designed to aid comprehension and scalability of the technique. The chapter first presents some motivation for the visualization. The chapter then presents the color metaphor that is used to present the suspiciousness and confidence metrics. The chapter next describes representations of the program being debugged at various levels of abstraction. The chapter then describes how the color applies to each representation level. The chapter next introduces a visualization of the test suite. Finally, the chapter describes how the different components of the visualization interact.

5.1 Motivation for Visualization

Chapter 3 presents the *suspiciousness* and *confidence* metrics that are used to provide a ranking of the coverage entities in the program to help guide the developers' inspection for the purposes of debugging. To make this information more accessible to the developer, we designed a visualization that represents these metrics and the program to be debugged. The suspiciousness and confidence values that are assigned to each coverage entity in the program can be difficult to interpret by the developer due to the possible large number of such entities. Also, the relationships among coverage entities with high suspiciousness values may be difficult for a developer to comprehend given only a large list of suspiciousness and confidence values. A ranking of coverage entities, from most suspicious to least suspicious, may also be difficult for a developer to interpret. For example, the developer may be presented with a listing of statement numbers that represents the ranking of those statements from

most suspicious to least suspicious. This listing would contain as many entries as there are statements in the program. The developer would be forced to find those statements in the program, one-by-one, without any context as to the suspiciousness values for other, related statements.

For these reasons, we created a number of visual metaphors that allow the Tarantula technique's results to be displayed to the developer in a way that (1) presents the suspiciousness and confidence values in an intuitive way, (2) scales to large programs, (3) enables a developer to interact and explore the fault-localization results, and (4) provides both a high-level, global view of the software as well as a low-level, local view.

5.2 Color Metaphors for the Suspiciousness and Confidence Metrics

Our visualization utilizes a continuous color (or hue) spectrum from red through yellow to green to color each coverage entity in the program under test. This color dimension maps to the suspiciousness metric. The intuition is that entities that are primarily executed by failed test cases (and are thus, highly suspicious of being faulty) are colored red to denote "danger;" entities that are executed primarily by passed test cases (and are thus, not likely to be faulty) are colored green to denote "safety;" and entities that are executed by a mixture of passed and failed test cases and thus, do not lend themselves to suspicion or safety, are colored in the yellowish range of colors between red and green to denote "caution." This color mapping uses a "traffic light analogy." We chose this as the default color mapping because people are already familiar with these colors and because of its intuitive meaning.

We use the "HSB" color model which describes color in terms of hue, saturation, and brightness. Of these three color dimensions, we use hue and brightness—the saturation is fixed at 100%. In particular, the hue of a coverage entity, e, is computed by the following equation:

$$hue(e) = 1 - suspiciousness(e) = \frac{\frac{passed(e)}{totalpassed}}{\frac{passed(e)}{totalpassed} + \frac{failed(e)}{totalfailed}} = \frac{\%passed(e)}{\%passed(e) + \%failed(e)} \tag{5.2.1}$$

In Equation 5.2.1, $passed(e)$, $failed(e)$, $totalpassed$, $totalfailed$, $\%passed(e)$, and $\%failed(e)$ have the same meaning as those given for Equation 3.3.1. We subtract the suspiciousness value (which varies from 0 to 1) from the value 1 because most computer color models place the red hue at 0 and the green hue at some value above that (often 0.33 or 33%).

The Tarantula tool uses the color model based on a spectrum from red through yellow to green. To achieve the range from 0 to 0.33 as many computer color models dictate for this spectrum, we multiply the result of Equation 5.2.1 by the value of 0.33 (or whatever the computer color model specifies for the green hue). However, the resulting $hue(e)$ can be scaled and shifted for other color models. The scaling or shifting may be useful for people that have different types of color vision deficiencies. The most common type of color vision deficiency affects the person's ability to differentiate red and green. Thus, for these people it would be useful to have a way to shift the spectrum.

In addition, another color dimension maps to the confidence metric. The confidence is visually encoded in the brightness of an entity. Bright entities represent high confidence and dark entities represent low confidence.

In particular, the brightness of a coverage entity, e, is computed by the following equation:

$$\begin{aligned} brightness(e) &= confidence(e) \\ &= max\left(\frac{passed(e)}{totalpassed}, \frac{failed(e)}{totalfailed}\right) = max\left(\frac{\%passed(e)}{100}, \frac{\%failed(e)}{100}\right) \end{aligned} \tag{5.2.2}$$

In Equation 5.2.2, the variables are the same as those defined above for Equation 5.2.1. Based on our experience and due to the limitations of computer displays and human perception, we also created a lower bound on the brightness value. Users often cannot perceive the difference between a pure black (brightness = 0) and a low value of brightness (e.g., brightness = 0.2. Thus, according to the limits of the users and the computer display, in practice we scaled the brightness as such

$$scaled\_brightness(e) = (range * brightness(e)) + (1 - range) \tag{5.2.3}$$

where *range* is defined as the range of perceptible brightness values. For example, if *range* were defined as 0.7, the brightness value would vary from 0.3 to 1.

5.3 Representation Levels

To investigate the program-execution data efficiently, the user must be able to view the data at different levels of detail. This visualization approach represents software systems at three different levels: statement level, file level, and system level.

We chose to focus attention on two-dimensional visualization techniques rather than three-dimensional techniques to minimize the interaction required by a user to see all dimensions of the display. With this approach, the user is not required to rotate the display to reveal obscured features as is often necessary with three-dimensional visualizations.

5.3.1 Statement Level

The lowest level of representation in the visualization is the statement level. At this level, the visualization represents source code, and each line of code is suitably colored (in cases where the information being represented does involve coloring). Figure 24 shows an example of a colored set of statements in this view. The statement level is the level at which users can get the most detail about the code. However, directly viewing

the code is not efficient for programs of non-trivial size. To alleviate this problem, the visualization approach provides representations at higher levels of abstraction.

```
...
finallyMethod.setName(
    handlers.getFinallyNameForCFGStartOffset(finallyStartOffsets[i] ));
if ( numFinallyBlocks != 0 ) {
    finallyMethod.setType(Primitive.valueOf(Primitive.VOID));
    finallyMethod.setContainingType(parentMethod.getContainingType());
}
finallyMethod.getContainingType().getProgram().addSymbol( finallyMethod );
finallyMethod.setDescriptor( new String("()V") );
finallyMethod.setSignature( parentMethod );
...
```

Figure 24: Example of statement-level view.

5.3.2 File Level

The representation at the file level provides a miniaturized view of the source code. This technique is similar to the one introduced by Eick and colleagues in the SeeSoft system [8, 27]: the technique maps each line in the source code to a short, horizontal line of pixels. Figure 25 shows an example of a file-level view. This "zoomed-away" perspective lets more of the software system be presented on one screen. Colors of the statements are still visible at this scale, and the relative colorings of many statements can be compared. This visualization represents each line of code with a line of pixels that is proportional to the length of the line of code, and the indentation is preserved. This approach presents the source code in a fashion that is intuitive and familiar because it has the same visual structure as the source code viewed in a text editor. This miniaturized view can display many statements at once. However, even for medium-size programs, significant scrolling is necessary to view the entire system. For example, the subject program for one of our feasibility studies, which consists of approximately 60,000 lines of code, requires several full screens to be represented with this view. Monitoring a program of this size would require scrolling back and

forth across the file-level view of the entire program, which may cause users to miss important details of the visualization. The scale limitations of this visualization motivates a higher level of abstraction, described in the next section.

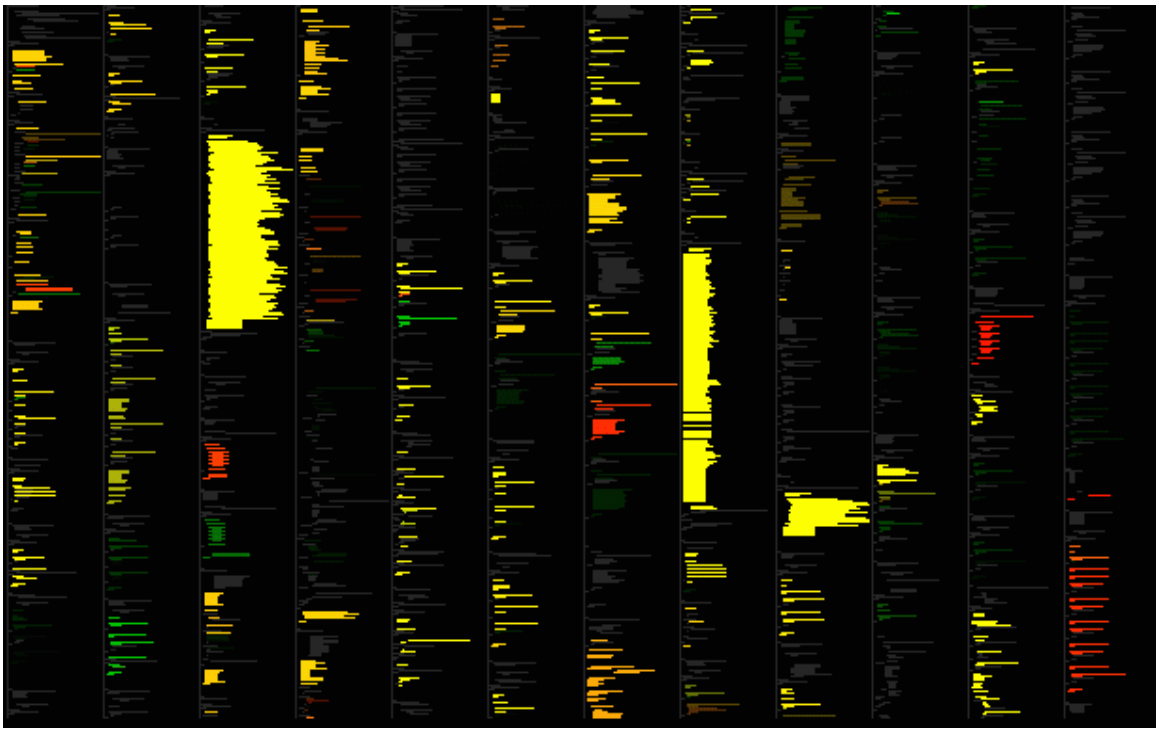

Figure 25: Example of file-level view.

5.3.3 System Level

The system level is the most abstracted level in my visualization. The representation at this level uses the treemap view developed by Shneiderman [69] as well as extensions to this view developed by Bruls and colleagues [14]. We chose to use treemaps because they are especially effective in letting users spot unusual patterns in the represented data and can scale to programs in the millions of lines of code.

In the development of the system-level view, we considered other visualization techniques such as Stasko and Zhang's Sunburst visualization [70] or Lamping, Rao, and Pirolli's Hyperbolic Tree visualization [46]. These techniques, however, focus

more on the hierarchical structure of the information they represent, and use a considerable amount of screen space to represent such structure. For our application, the hierarchical structure of the program modules is less important than representing as much information as possible at each level of the hierarchy. With treemaps, the entire screen space can be used to represent the color information for the hierarchical level being considered (e.g., a package or the classes in a package) without using valuable screen space to encode redundant information about nodes' parents. The hierarchical structure is used only to group nodes belonging to common branches of the tree.

For the system-level view, the tool builds a tree structure that represents the system. The root node represents the entire system. The intermediate non-leaf nodes represent modularizations of the system (e.g., Java packages). The leaf nodes represent source files in the system. The treemap visualization is then applied to this tree. The size of the leaf nodes is proportional to the number of executable statements in the source file that it represents.

5.4 Coloring for Different Representation Levels

Each statement in the program is assigned a color according to the hue and brightness variables defined previously, but the coloring applies differently to the different visual representation levels. For the statement-level and the file-level representations, no mapping is necessary: for each statement, the color (i.e., hue and brightness) of the statement is used to color the corresponding line of code in the statement-level representation and the corresponding line of pixels in the file-level representation.

For the system-level representation, there is no one-to-one mapping between statements and visual entities. Therefore, we defined a mapping that maintains color-related information in the treemap view. Each leaf node (i.e., rectangle) in the treemap view represents a source file.

To map the color distribution of the statements in a source file to the coloring of

the node that represents that source file, we developed a treemap-like representation to further partition each node (in this sense, we are embedding a treemap within each treemap node). For example, if half the statements in a source file were colored bright red, and the other half were colored dark green, the treemap node would be colored as such—half of it would be colored bright red and half of it would be colored dark green.

Using a traditional treemap algorithm for coloring the nodes would likely cause the colors to be laid out in a different fashion for different nodes. For example, suppose the colors assigned to the statements in source file A were evenly distributed among four colors: bright red, dark red, bright green, and dark green. To color the node in the treemap view, we may use a traditional treemap algorithm to further divide node A (that represents source file A) into four equally-sized blocks, each colored by one of the specified colors. However, in a traditional treemap algorithm, relative placement of nodes is not guaranteed. So, in node A, the bright red block may be placed in the upper-right corner, but in node B, which represents similar proportions of colored statements, the bright red block may be placed in the lower-left corner. In a treemap view that contains many nodes, a non-uniform appearance of the nodes will likely cause confusion as to where the boundaries of the nodes lie. Therefore, we chose to keep the same layout of colors within each node while still showing the color distribution in a treemap-like fashion. The layout is characterized by varying the hue across the horizontal axis and by varying the brightness across the vertical axis. Figure 26(b) shows an example of this layout, where hue ranges from green, through yellow, to red on the horizontal axis, and the brightness varies from dark to bright on the vertical axis.

This layout determines the relative placement of the colors within each treemap node, but does not define how the colors are mapped to colors assigned in the statement-level or file-level representations. Thus, we defined a technique for skewing

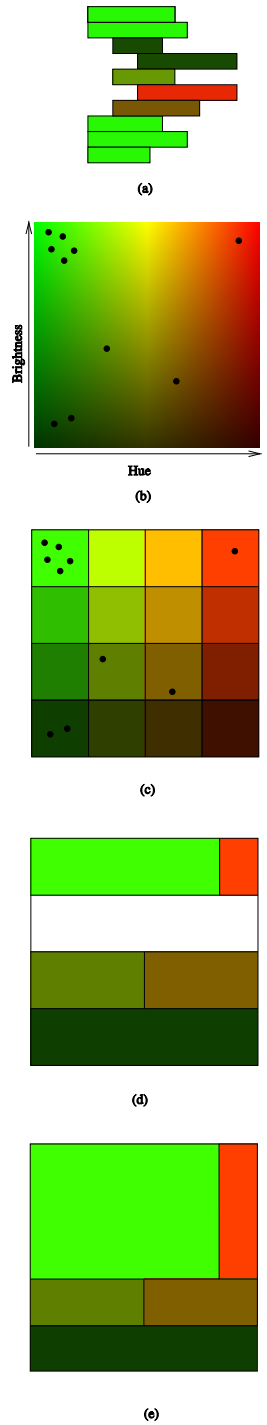

Figure 26: Example that illustrates the steps of the treemap node drawing.

the colors of Figure 26(b) to present the appropriate proportions of colors assigned while preserving the layout of the colors.

I will explain this technique while illustrating it on the example in Figure 26. Assume that the sample file-level view shown in Figure 26(a) is a source file composed of a set of statements, with related colorings, to be mapped into a treemap node. The skewing of the color layout is performed in four steps. The first step plots the color of each statement onto a coordinate system with hue varying across the horizontal axis and brightness varying across the vertical axis. For the example, this step would result in the points plotted on the hue/brightness space in Figure 26(b), in which each point represents a statement in Figure 26(a) positioned at the appropriate hue and brightness.

The second step segments the space horizontally and vertically into equal-sized blocks to create a discrete bucket for each block, so as to categorize the statements' colors. This segmentation is shown in Figure 26(c). For the sake of simplicity, this example uses only four segments vertically and four segments horizontally, resulting in sixteen blocks; however, in a real application, this could be tuned to a finer-grained categorization. After the segmentation is complete, each block is drawn with a representative color.

The third step determines, for each row, the width of each block. To this end, the technique computes the ratio of the number of statements in the block to the number of statements in the entire row. The width of each block is proportional to this ratio. The widths of the blocks for the example are shown in Figure 26(d). The technique assigns the leftmost block in the first row 5/6th of the total width of the node because five of the six points in the row fall into this block. Likewise, the coloring technique assigns the rightmost block the remaining 1/6th of the width of the node. The middle two blocks in the first row are eliminated (i.e., they are assigned width 0) because they contain no points. Note that the technique assigns no widths for the second row

because no points fall into this row.

The final step determines the height of each row by computing the ratio of the number of statements in the row to the number of statements in the entire node. The heights of the blocks for the example are shown in Figure 26(e), which is the final representation of the node. The technique assigns the first row 6/10th of the total height of the node because six of the ten points in the node fall into this row. The last two rows are each assigned 2/10ths of the total height of the node.

This coloring technique results in blocks that are proportional in size to the number of statements plotted in them and, in addition, maintains the layout of the color blocks for each node. For example, the brightest green block, which contained five of the ten statements, results in half of the total area of the node ($5/6 * 6/10 = 1/2$).

5.5 Representation of Executions

To represent executions, we defined an *execution bar*: a rectangular bar, of which only a subset is visible at any time. The bar consists of bands of the same height of the bar but of minimal width. *Minimal width* refers to a width that is as little as possible but can still be seen. The actual width depends on the characteristics of the graphical environment, such as the size and resolution of the display. Figure 27 shows a simple example of an execution bar.

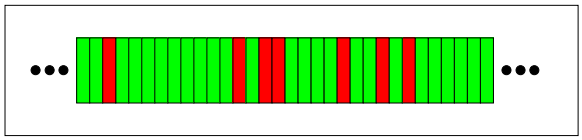

Figure 27: Example of execution bar.

Each band in the execution bar represents a different execution of the program and is colored according to the pass/fail status of the test case that it represents: green for a passed test case and red for a failed test case.

5.6 Integration of Visual Components

Visual components interact with each other to let the user navigate and explore the information displayed. For example, if the user selects one or any subset of test cases in the execution bar, the other views update with the results of Tarantula performed on only those test cases. If the user clicks on any of the components in any of the views, the other components focus on that component.

The integration and layout of these component is shown in the screen capture presented in Figure 28. In this figure, the statement-level view is in the lower left, the file-level view is in the center-left, and the system-level view is in the center-right. In addition to these major components, the tool contains components for convenience and informational purposes. These include an interactive color legend, a statistics pane, a color slider, and menus to control the color-space.

First, in the lower right of Figure 28 is an interactive color legend. The color legend is drawn as a two-dimensional plane with hue varying on the horizontal axis and brightness varying on the vertical axis. The color legend includes a small black dot at each position in the color space occupied by a source-code statement. By selecting a bounding rectangle in this region around some points, the user can modify (filter) the main display area, showing and coloring only statements having the selected color and brightness values.

Second, immediately above the interactive color legend in Figure 28 is the statistics pane. This pane shows some information about the last statement that was moused-over in the file-level or source-level views. For example, in Figure 28, the mouse was last placed over line 1066 of `jaba/graph/cfg/CFGImpl.java`. This statement was executed by 90 of the 707 test cases. Twenty four of the 707 test cases failed, and 20 of them executed this statement.

Third, in the upper left corner of the interface in Figure 28 is a slider that controls the grayscale brightness of lines not being drawn using the red-yellow-green mapping

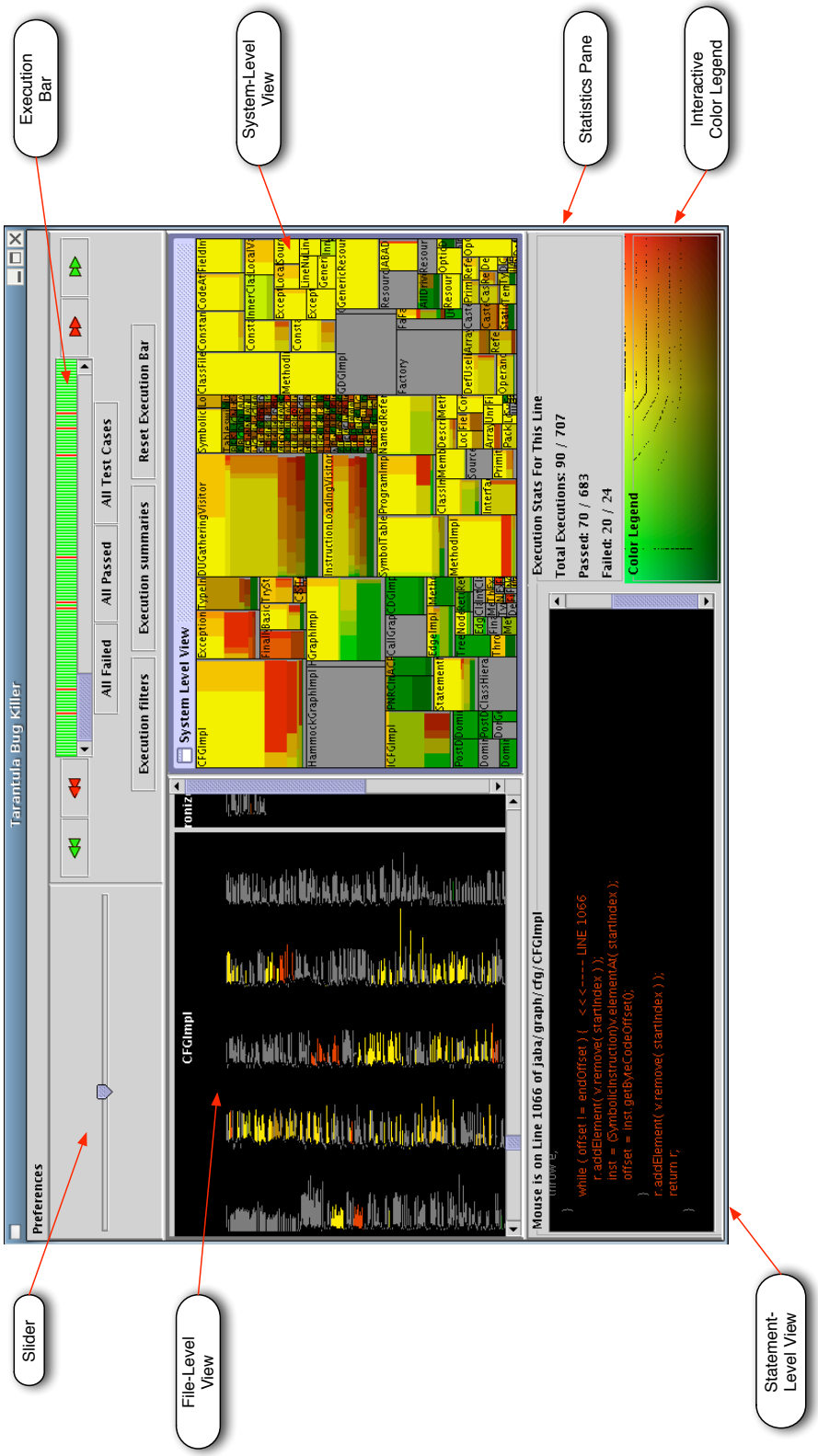

Figure 28: Screenshot of the Tarantula tool.

(comments, unexecuted lines, filtered lines, etc.). If the slider is moved to the left, the unexecuted and unexecutable statements become darker, letting the user focus only on the executed statements. If the slider is moved to the right, the unexecuted and unexecutable statements become brighter, letting the user see the full structure of the code. In Figure 28, the slider is positioned to show those statements not involved in the mapping as light gray.

Fourth, in the upper right corner, below the execution bar, are some controls to filter the test cases that are used. These controls provide convenient access to explore how different subsets of the test suite affect the Tarantula results.

And finally, under the "Preferences" menu are controls to change the color-space used. This may be especially useful for color-blind users.

CHAPTER VI

IMPLEMENTATION OF THE TARANTULA SYSTEM

To evaluate the techniques presented in this dissertation, we implemented a number of tools. This chapter presents the tools that were developed to enable the techniques described in the previous chapters. First, this chapter provides an overview of the implementation. Then, it presents the tools that are used to provide dynamic execution information about the test cases. Next, it describes the implementations for the clustering of failed test cases. The chapter then describes the components that were developed to enable the visualization of the fault-localization results. The chapter next describes the ranking that was implemented for the purposes of evaluation. Finally, the chapter describes how the fault-localization techniques were implemented in a way that enabled the localization of faults in deployed software.

6.1 Implementing Tarantula: An Overview

The Tarantula implementations were primarily written in Java with various supporting tools and scaffolding written in a variety of languages such as Perl, Python, Bash, and C#. Figure 29 shows a high-level data flow diagram of the Tarantula System. Figure 30 provides a legend for the visual components used in Figure 29. The system is composed of a number of processes. First, the program is instrumented so that it produces dynamic coverage information upon execution. This program is either executed with the test suite, or deployed to clients. The Gammatella subsystem of Tarantula handles the deployment and capture of the data from the clients in the field. Regardless of whether the dynamic information comes from the in-house test suite, or the in-the-field client executions, the information is collected and processed. The execution information can be sent directly to the component that computes the

suspiciousness and confidence metrics, or it can be first sent to the clustering subsystem. For the clustering subsystem, the failed test cases are input to the clustering tool, which outputs the clusters of failed test cases, i.e. fault-focusing clusters. These fault-focusing clusters along with the passed test cases are used to create specialized test suites. The specialized test suites are used to compute the suspiciousness and confidence metrics. The values of these metrics are used to either compute the ranking of the coverage entities (for use in experimentation) or to generate visualizations of the results (for use by developers). The components of this diagram will be described in the next section.

6.2 Instrumenting and Coverage Processing

The prototype implementations of Tarantula have used instrumentation to provide statement coverage. We integrated it with a few different instrumentation tools. We integrated it with the statement-coverage instrumenter that is built into the Aristotle Analysis System [6]. The Aristotle Analysis System provides both static- and dynamic-analysis tools for programs written in C. It uses the Edison Design Group's (EDG) C front-end compiler [26] to parse the program and produce syntax trees. The instrumenter, `il-st`, also uses the EDG tools to provide support for program modification. This instrumenter transforms the syntax tree that EDG produces to include basic-block level probes. The instrumenter then utilizes functionality built-into the EDG tool that permits source code to be generated from the syntax tree. The resulting generated source code is then compiled and executed by the test suite. Each execution of the program produces a new coverage file that represents which entities were executed during that execution. All of the coverage files for a test suite are then input and parsed by the Tarantula tool.

Another version of the Tarantula tool uses instrumentation that was provided by

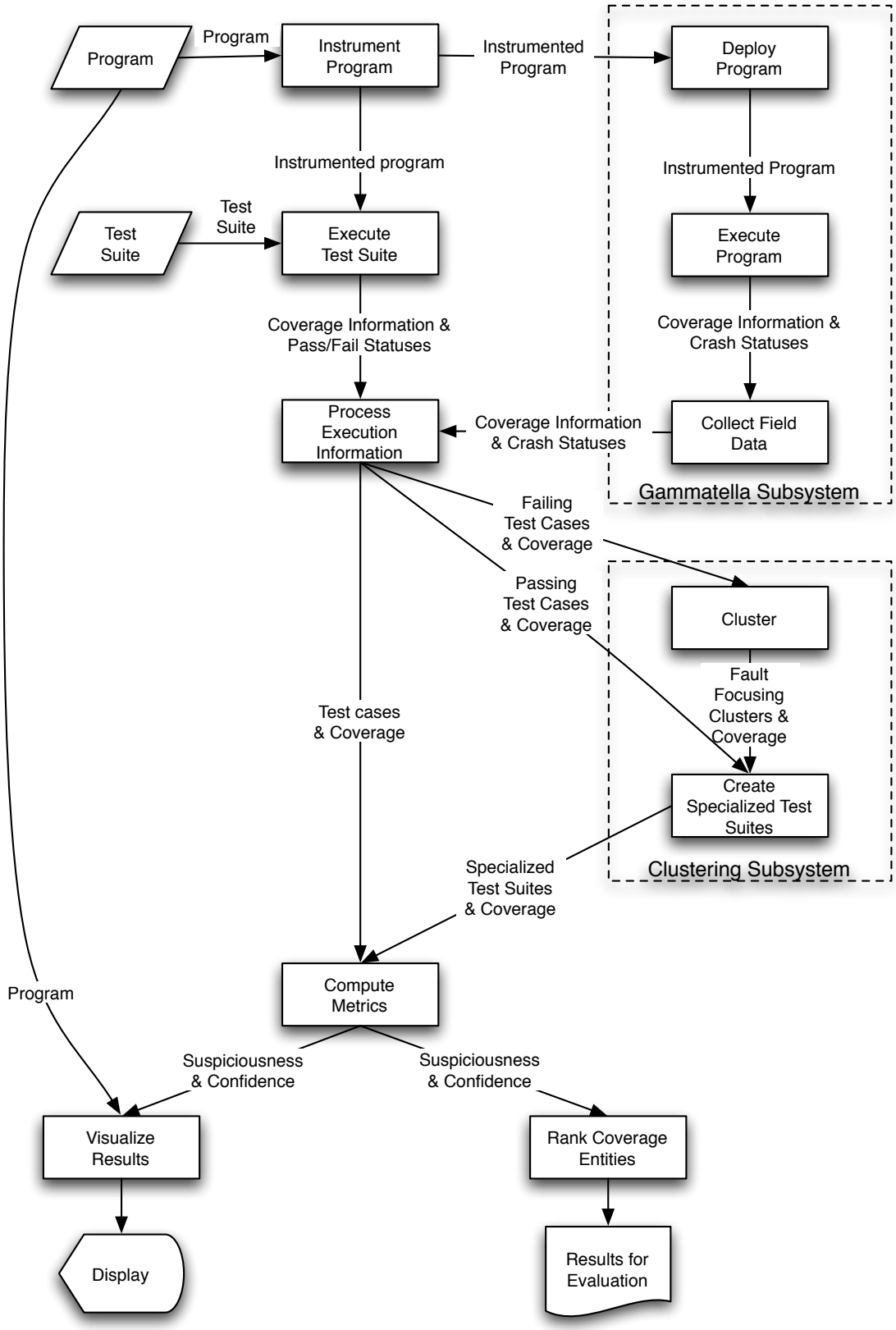

Figure 29: Diagram of the Tarantula System Implementation

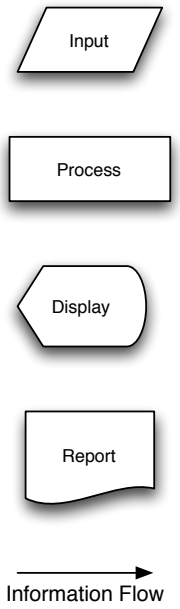

Figure 30: Legend for the Dataflow diagram in Figure 29.

the GNU C compiler, gcc [30]. The gcc tool provides functionality that lets it insert coverage probes into the binary machine code. When gcc is provided with the "`-ftest-coverage -fprofile-arcs`" arguments, it compiles the source code into an executable file that also has the coverage probes included. When the executable file is executed, it records which statements and branches were executed and the number of times each was executed during that execution. After the execution terminates, the profiling information is output to a binary, machine-readable file. The instrumentation provides cumulative profiling information—that is, the profiling represents the sum of all executions thus far. To capture the dynamic information for each execution, individually, the profiling-output information is copied, and the execution profile counters are then reset to zero. Another supporting tool, gcov [31], that is included in the gcc software package, enables the reading of the files that contain the profiling information. The gcov tool reports for each statement in the program one

of three possible results: (1) the statement is unexecutable (for example, it is a comment, variable declaration, or blank line); (2) the statement is executable, but it was not executed; or (3) the statement was executed n times. When parsing the output of gcov, Tarantula translates the profiling information to coverage information: if a statement is executed n times and $n > 0$ it reports that that statement *was executed*.

Another version of the Tarantula tool uses instrumentation that was provided by the InsECT (for Instrumentation, Execution, and Coverage Tool) instrumenter [16]. InsECT instruments programs that are written in Java. It takes the compiled byte-code file as input, inserts probes, and outputs a modifies byte-code file that can be executed by the Java virtual machine. InsECT is capable of providing instrumentation of various coverage entities. Tarantula uses the statement-level instrumentation capabilities in InsECT. Each execution of the program produces a new coverage file that represents which entities were executed during that execution. InsECT provides a Java-language library that can be linked and used that lets the client program, Tarantula in this case, query the execution coverage information for the program under test.

6.3 Clustering of Failed Test Cases

We implemented the clustering of failed test case to enable multiple-fault debugging and parallel debugging by integrating three existing systems: the Aristotle Analysis System, the Argo execution clustering tool, and the Tarantula Debugging Tool. The Aristotle Analysis System [6] is written in C and runs on Solaris. Among other things, it is capable of analyzing and instrumenting C programs. The instrumenter that we used provides branch profiles of executions. Upon running the test suite, the profiles are saved to disk and sent to the Argo clustering tool.

Argo [11] is written in C# and runs on the .Net virtual machine in Microsoft Windows. Argo models and clusters program executions using the approach described

in Section 4.2.1. Argo takes the execution branch profiles of the failed test cases as input and outputs a dendrogram. The dendrogram is saved and sent to the fault-localization module for the determination of the stopping point for the clustering and the refinement of the clusters.

The fault-localization module was written in Java and was run on Linux. It computed the fault-localization results for every node in the dendrogram to determine the stopping point for the clustering, as described in Section 4.2.1.2. Once the stopping point for the clustering was determined, the fault-localization module was used to refine the clusters, as described in Section 4.2.1.3.

We also wrote cross-platform scaffolding to support the coordination of the multiple programs across multiple operating systems and computers. The scaffolding provided communication among the Aristotle Analysis System, the Argo system, and the Tarantula fault-localization module across the Linux, Microsoft Windows, and Solaris platforms. The scaffolding also provided the results processing and the central guidance of how the processing should continue for derivative multi-fault versions of the subject program. The scaffolding was built using Unix shell scripting and the Perl programming language.

6.4 Computing Fault-Localization Metrics

The computation of the *suspiciousness* and *confidence* metrics was written in Java. For every coverage entity that was monitored, it computes the metrics, as was described in Section 3.3. Each specialized test suite is input along with the coverage information gathered from the instrumented program. The coverage information is parsed and the coverage matrix is built. For each of the specialized test suites, the *suspiciousness* and *confidence* values for each coverage entity is output. The computation module was also used to experiment with other metrics.

6.5 Visualizing Fault-Localization Results

The visualizer is the module of Tarantula that implements the visualization techniques described in Chapter 5. The visualizer is divided into several interacting components that are all written in Java using the graphical capabilities of the Swing toolkit. For the Treemap view, we modified and extended Bouthier's publicly available Treemap Library [10].

6.6 Ranking the Coverage Entities

The ranking of the coverage entities is done by sorting the coverage entities that were instrumented. The sorting occurs as was described in Section 3.4. The suspiciousness metric is used as the primary sorting key and the confidence metric is used as the secondary sorting key (breaking any ties among entities with equivalent suspiciousness values). This ranking computation was written in Java. It is used to inform the evaluation of the Tarantula technique, which will be described in Chapter 7.

6.7 Monitoring and Debugging Deployed Software

One application of Tarantula that we explored was its use in monitoring deployed software. We called the version of Tarantula that was applied to deployed software "Gammatella." To enable the monitoring and fault localization of software after it is deployed to clients, we implemented the Tarantula technique in a way that can provide communication between the client software and the developers. The program was instrumented using the InsECT tool that was described in Section 6.2. At the end of an execution at the client's site, or at given time intervals (e.g., in the case of continuously running applications), the information is dumped, compressed, and sent back to a central server over the network.

We use the SMTP protocol [58] to transfer the program-execution data from the clients' machines to the central server collecting them. The compressed data are

attached to a regular electronic-mail message whose recipient is a special user on the server and whose subject contains a given label and an alphanumeric ID that uniquely identifies both the program that sent the data and its version.

6.7.1 Data Collection Daemon

The *Data Collection Daemon* is a simple tool written in Java that runs as a daemon process on a server on which we store the execution data. Each instance of the tool monitors for execution data from all instances of a specific version of a specific program, provided to the tool in the form of the corresponding alphanumeric identifier. The tool, upon execution, retrieves the incoming mail for the collection user from the server. To facilitate access of the data from different machines, we use the Internet Message Access Protocol (IMAP [72]).

For each message retrieved, the daemon parses the subject of the message to check whether (1) the message contains coverage information (i.e., the subject contains the coverage label) and (2) the information is coming from the correct program and version (i.e., the ID provided to the daemon matches the one in the subject). If both conditions are satisfied, the daemon extracts the attachment from the message, uncompresses it, and suitably stores the program-execution data in a database. The additional information about each execution, such as the Java Virtual Machine version and the user ID, are stored as properties of the execution.

6.7.2 Public Display of Gammatella

To enable developers to learn about failures in the field for their clients and to explore the parts of the programs related to those failures, we envisioned a display of the Tarantula visualization that would be placed in a public place. Here, developers could interact with the visualization to investigate possible causes of failures. Because the display would be placed in a public area so that all developers could see it, developers may be prompted to interact with each other, as well, to discuss problematic parts

of the program and how the program is being used by their clients. A view of a prototype public display is pictured in Figure 31.

Figure 31: Public display of GAMMATELLA.

CHAPTER VII

EXPERIMENTATION

To validate my thesis, we performed a number of studies. Section 7.2 presents a study that demonstrates the effectiveness of the Tarantula technique by examining the suspiciousness values that are assigned to faulty and non-faulty statements. Section 7.3 presents a study that demonstrates the relative effectiveness of the Tarantula technique by comparing it with a number of other fault localization techniques. Section 7.4 presents a study that demonstrates the effectiveness of the Tarantula technique for programs with multiple faults, both with and without the aid of the partitioning of test suites, using the clustering techniques presented in Chapter 4. Section 7.5 presents a study that demonstrates the effectiveness of debugging in parallel. Section 7.6 demonstrates the efficiency of the Tarantula technique by presenting timings of applying the Tarantula technique. Section 7.7 demonstrates the efficiency of the clustering techniques used to enable creation of specialized test suites and debugging in parallel. Finally, Section 7.8 examines the effects of test suite composition on the effectiveness of fault localization techniques that use dynamic testing information.

7.1 Subject Programs

Table 2 lists the subject programs that were used for these studies. It also shows the number of faulty versions, the averages of the numbers of lines of code (LOC) in each program across all versions, the numbers of test cases in each test pool, and descriptions of the functionalities of the programs.

Table 2: Subject Programs.

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
`tcas`	41	141	1578	altitude separation
`tot_info`	23	440	1054	information measure
`space`	38	6218	13585	array definition interpreter
`space` (8-fault versions)	100	6218	13585	array definition interpreter

7.1.1 Siemens Suite

The Siemens programs, 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 [37]. The suite consists of seven programs: `print_tokens`, `print_tokens2`, `replace`, `schedule`, `schedule2`, `tcas`, and `tot_info`. The `print_tokens` and `print_tokens2` programs are lexical analyzers. The `replace` program performs pattern matching and substitution. The `schedule` and `schedule2` programs are schedulers. The `tcas` program models an aircraft collision avoidance algorithm. Finally, the `tot_info` program computes statistics [63].

The researchers at Siemens created test cases for these programs. They first created black-box tests "according to good testing practices, based on the tester's understanding of the program's functionality and knowledge of the programs functionality and knowledge of special values and boundary points that are easily observable in the code" [37, p. 194]. To do this, they used the category partition method and their Siemens Test Specification Language tool [7, 54]. In addition, they created white-box tests so that every statement, edge, and definition-use pair was covered by at least 30 test cases.

The researchers at Siemens also created faulty versions of the programs. They

modified between one and five lines of code to introduce faults into the programs. Their goal was to introduce realistic faults based on their experience. To accomplish this goal, the researchers retained only faults that were neither too easy nor too hard to detect [37, p.196], which they defined as being detectable by at most 350 and at least three test cases in the test pool associated with each program.

7.1.2 `Space` Program

The `space` program functions as an interpreter for an array definition language (ADL). The program reads a file that contains several ADL statements. It checks the contents of the file for adherence to the ADL grammar and to specific consistency rules. If the ADL file is correct, the `space` program outputs an array data file containing a list of array elements, positions, and excitations; otherwise the program outputs error messages. `Space` consists of 6218 executable lines of C code. We also used 38 faulty versions of the program, each containing one fault, and the *base* version, which is assumed for purposes of the studies to contain no faults. Thirty-three of the faulty versions, each contain a single fault that had been discovered during the program's development. Through working with `space`, Rothermel and colleagues [65] discovered an additional five faults and created versions with those faults. In addition, we created a number of multiple-fault versions of `space` by isolating the faults found in each of the 38 other versions and injecting them into the base version. The method for doing this injection of multiple faults is explained in the studies that use them, in Sections 7.2, 7.4, 7.5, and 7.8

The test suite for `space` was constructed from 10,000 test cases generated randomly by Vokolos and Frankl [74], and then 3,585 test cases were created by researchers in the Aristotle Research Group to guarantee that each executable edge in the program's CFG was exercised by at least 30 test cases [63].

7.2 Study 1: Evaluating the Effectiveness of the Tarantula Technique by Examining the Accuracy of the Suspiciousness Metric

To evaluate the effectiveness of the Tarantula technique, we studied the accuracy of the suspiciousness metric. This study examines the value of the suspiciousness assigned to both faulty and non-faulty statements. By examining both faulty and non-faulty statements, we assess the frequency that the technique correctly assigns high suspiciousness values to faulty statements and the frequency that the technique incorrectly assigns high suspiciousness values to non-faulty statements. We performed this study for both single-fault versions of the subject program and multiple-fault versions to study whether the number of faults had any effect on the technique. The results showed that the technique successfully narrows the search space for the fault by assigning high suspiciousness values to a large percentage of faulty statements and assigning high suspiciousness values to a small percentage of non-faulty statements. The results also showed that the technique was generally less effective for programs with multiple faults.

7.2.1 Object of Analysis

For this study, we used the `space` program that was described in Section 7.1. Using the 13585 test cases in the overall test pool of test cases, we extracted 1000 randomly sized, randomly generated, near-decision-adequate test suites from this test pool. This subject and these test suites have been used in similar studies (e.g., [28, 41, 64, 66]). These test suites are near decision-coverage-adequate: they covered 80.7% to 81.6% of the 539 conditions in the 489 decisions. The test suites ranged in size from 159 to 4712 test cases.

We used 20 of the single-fault versions for the `space` program. We chose these 20 versions because they were the versions for which there was at least one failed test

case in the 1000 test suites that we generated. We also randomly generated 10 two-fault versions, 10 three-fault versions, 10 four-fault versions, and 10 five-fault versions of the `space` program.

7.2.2 Variables and Measures

We computed the suspiciousness values for each statement for all test suites, and for all faulty versions of the subject program.

Our experiment manipulated one independent variable: the faulty version of the `space` program. The study considered 20 single-fault, 10 one-fault, 10 two-fault, 10 three-fault, 10 four-fault, and 10 five-fault versions of `space`.

To assess the accuracy of the Tarantula technique, we used one dependent variable: the suspiciousness value. The suspiciousness value was computed according to the metric described in Section 3.3.

7.2.3 Experimental Setup

We executed each version with instrumentation on each of the 1000 test suites, and then applied the Tarantula technique to each version-test suite pair. The instrumentation gathered the coverage information about the test suites. We computed the suspiciousness value for each of the faulty and non-faulty statements in the program.

7.2.4 Results and Discussion

To determine how frequently the technique assigns an appropriately high suspiciousness value to faulty statements, we examine the suspiciousness values assigned to all faulty statements. Figure 33 shows the results of this part of the study as a segmented bar chart. The chart contains one bar for each of the twenty versions of the program that we studied. Each bar in the segmented bar chart represents 100% of the faulty statements of that version of the program across all test suites. Each segment is color coded to represent the range of suspiciousness values that it represents. Figure 32

is a legend that creates the mapping from suspiciousness value ranges to the color representations of the segments. Each of these representative colors that is used in the segmented bar charts is the mean hue of the colors that would have been used if they were to be colored according to the coloring described in Section 5.2 for the suspiciousness values in that range.

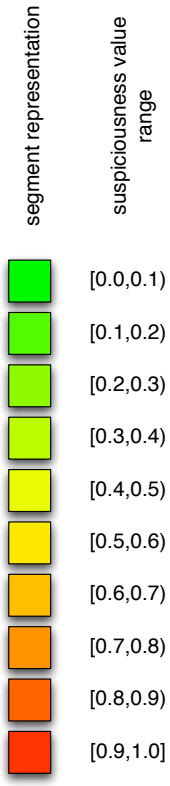

Figure 32: Legend for Figures 33, 34, and 35 mapping suspiciousness value ranges to representational colors.

Each segment of each bar represents the number of times that the faulty statements were assigned suspiciousness values in the represented range. The size of each segment represents the percentage of the times that the faulty statements were assigned suspiciousness values in that range across all test suites. For example in Figure 33, across all 1000 test suites, version 20 (the rightmost bar in the chart) had approximately 3% of its faulty statements with suspiciousness values between 0.9 and

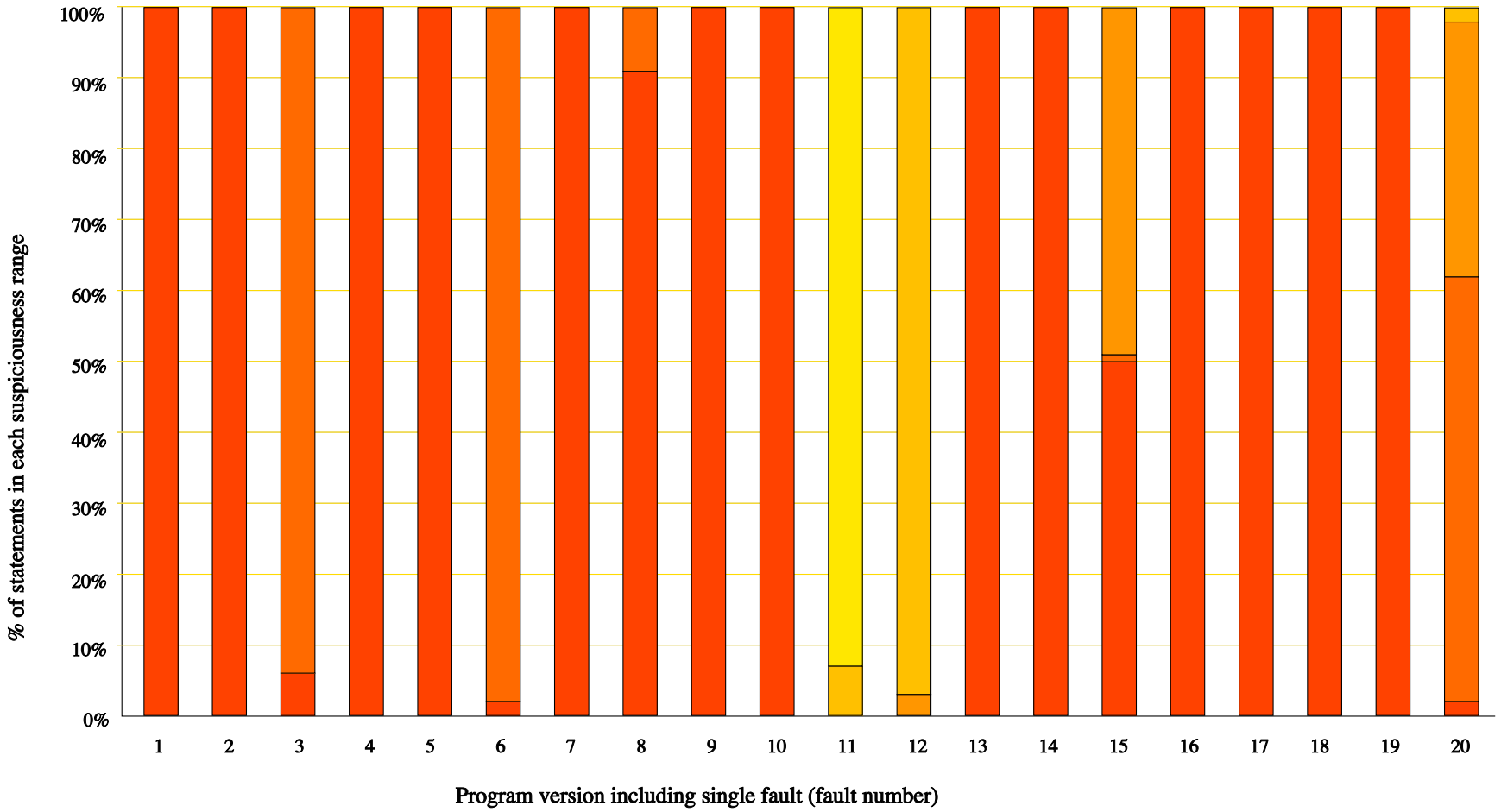

Figure 33: Resulting suspiciousness values assigned to all faulty statements across all 1000 test suites for 20 versions of `space`.

1.0, 60% with values between 0.8 and 0.9, 34% with values between 0.7 and 0.6, and 3% with values between 0.5 and 0.6.

Figure 33 shows that, for our program and versions, most of the faulty statements across all 1000 test suites were assigned suspiciousness values in the three most suspiciousness ranges—greater than 0.7. However, for two versions—11 and 12—the faulty statements were assigned suspiciousness values between 0.5 and 0.8. We examined these two versions, and discovered that in them, the fault was in code that initializes variables in statements that are executed by all or most test cases. For these versions, the fault manifests itself as a failure later in the code.

To determine how frequently the technique assigns an appropriately low suspiciousness value to non-faulty statements, we examined the suspiciousness values assigned to all non-faulty statements. For this part of the study, we applied the same technique, and we display our results in the same fashion as in Figure 33 except that the segmented bar chart represents the non-faulty statements, instead of the faulty ones.

Figure 34 shows these results. In all 20 versions, less than 20% of the non-faulty statements are assigned suspiciousness values above 0.8, and often much less, indicating that, for this subject, faults, and test cases, the technique significantly narrows the search space for the faults. Of the statements with suspiciousness values above 0.8, we found that most of these statements immediately surround the fault in terms of code listings. For example, if the statements immediately preceding and following the fault in the code listing are also assigned high suspiciousness values, the technique would still draw the developer's attention to the faulty area of the code.

It is worth noting that versions 11 and 12, whose faults were assigned suspiciousness values between 0.5 and 0.8, have almost no highly suspicious faulty or non-faulty statements. This means that for these versions, the technique does not mislead the user, but simply fails to highlight the fault—no or few false positives.

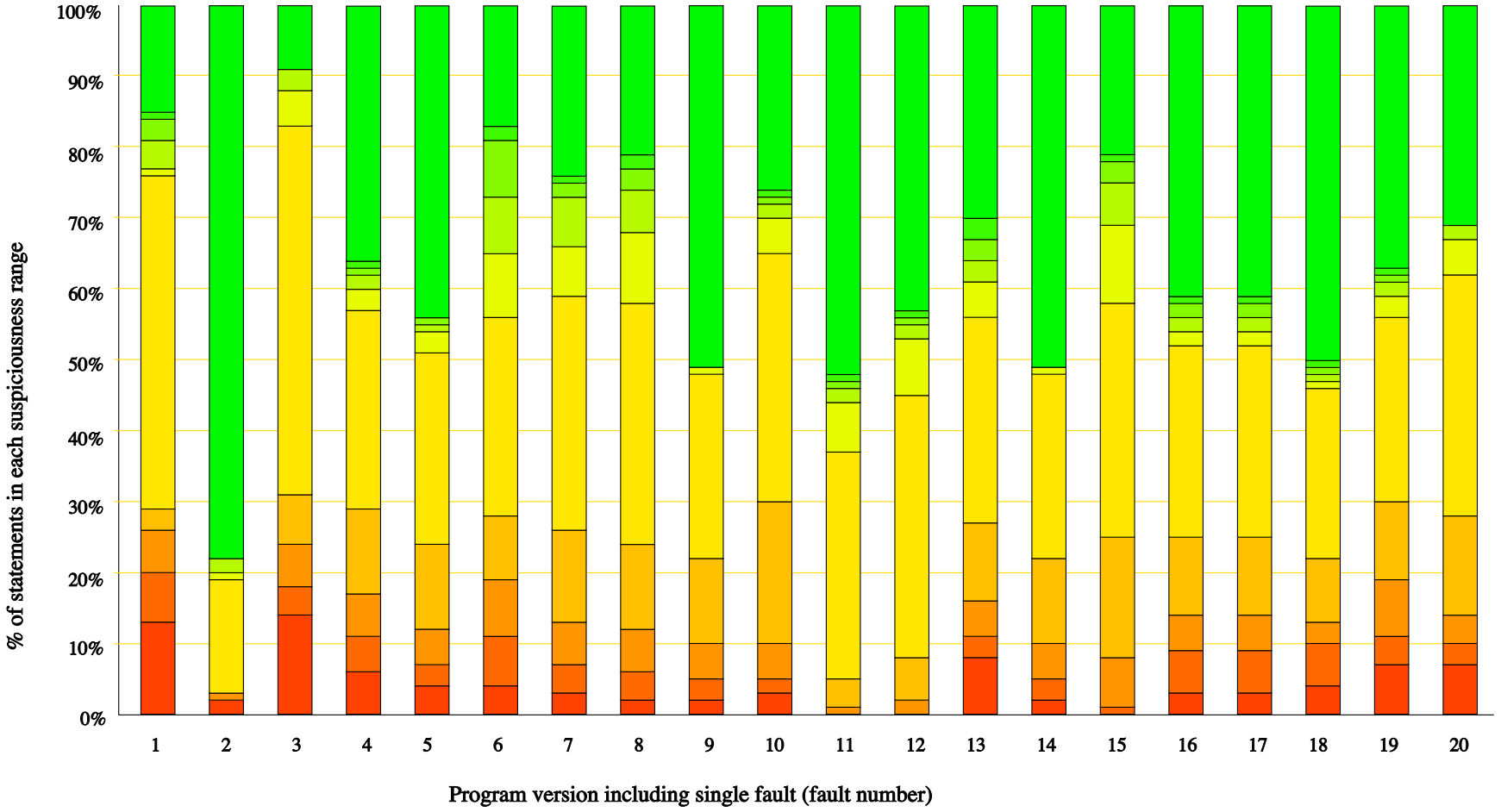

Figure 34: Resulting suspiciousness values assigned to all non-faulty statements across all 1000 test suites for 20 versions of `space`.

To examine how the technique performs for programs with multiple faults, we examined the suspiciousness values assigned to all faulty statements for several multiple-fault versions of `space`. Figures 35(a)-35(d) show the results of this part of the study in the same segmented bar-chart manner as in Figures 33 and 34. As expected, the effectiveness of the technique declines on all faults as the number of faults increases. However, even when there are up to five faults, a large majority of the faults are assigned suspiciousness values greater than 0.5. In fact, a large portion of the faulty statements are assigned suspiciousness values greater than 0.7. These charts show that the percentage of faulty statements with low suspiciousness values is greater than those seen for the single-fault versions in Figure 33. Overall, for the results shown in Figure 35, the decline in effectiveness in highlighting the faulty statements is less than we expected. Even up to five faults, the technique performed fairly well.

We expected that the results of this study may be somewhat misleading. Because we are presenting the number of faulty statements in each segment for *all* faults in Figures 35(a)-35(d), suspiciousness values of individual faults are not distinguished. For example, the second bar of Figure 35(a) does not allow us to determine how the individual faults (14 and 17) fell into the different ranges of suspiciousness values. Were both fault 14 and fault 17 assigned high and low suspiciousness values? Or, were all of the low suspiciousness values assigned to one of the faults? We believe that this distinction is important because a fault that is not illuminated by the technique may eventually be illuminated if another more evident fault is located and removed. To investigate this situation, we plotted the data for each individual fault of a multi-fault version. From the left side of Figure 35, we chose a segmented bar that had both high and low suspiciousness value ranges to dissect.

For this case study, we chose the two-fault version containing faults 14 and 17—the second bar in Figure 35(a). Figure 36 shows the results of this case study. The first segmented bar in Figure 36 shows the suspiciousness values assigned to the statement

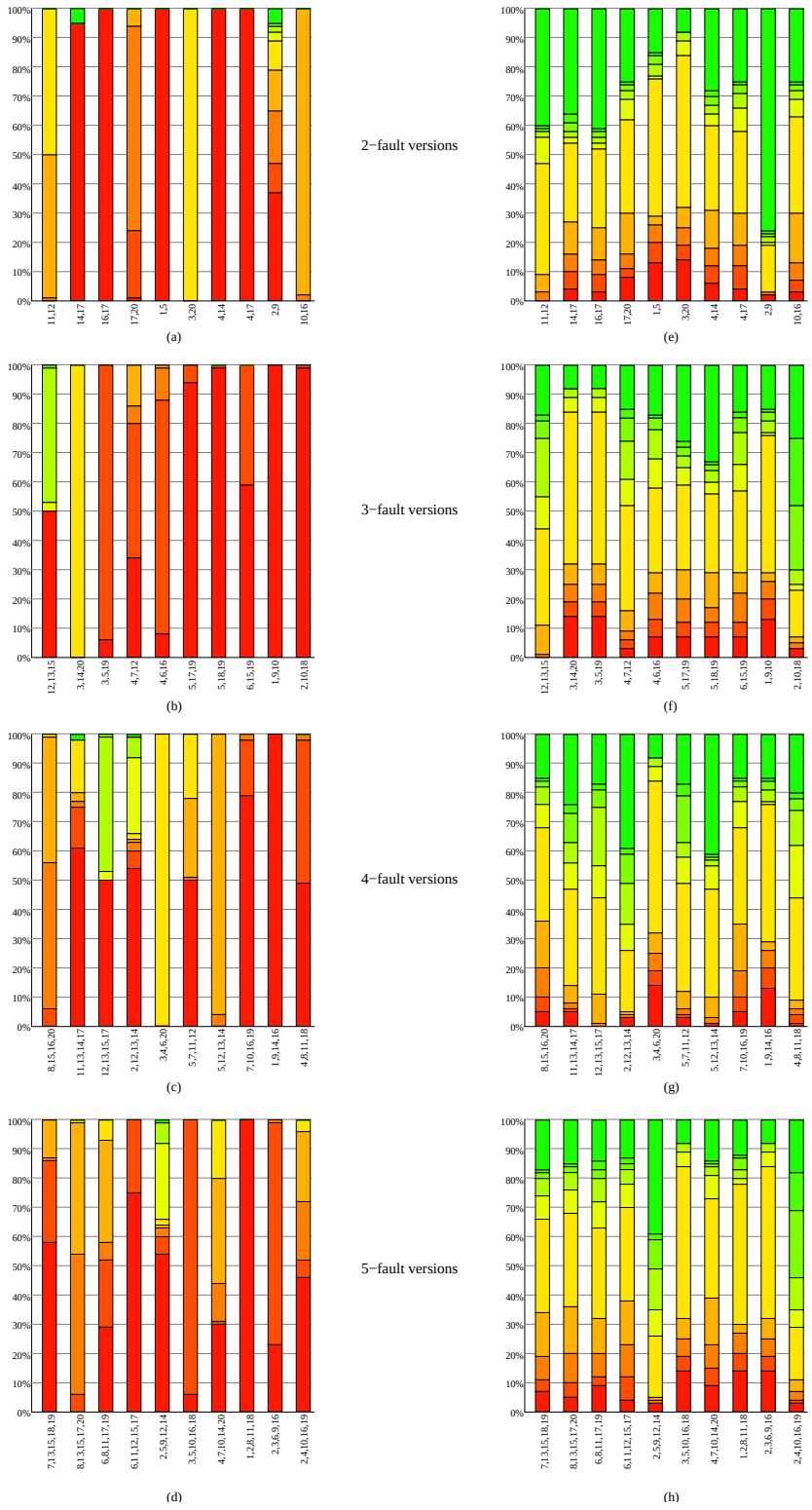

Figure 35: Resulting suspiciousness values for the faulty statements (left) and non-faulty statements (right) in multiple fault versions across all test suites.

containing fault 14 across all 1000 test suites. The second segmented bar shows the suspiciousness values assigned to the statement containing fault 17 across all 1000 test suites. Fault 14 is assigned a suspiciousness value greater than 0.9 for 90% of the test suites, between 0.8 and 0.7 for 1% of the test suites, and between 0.0 and 0.1 for 10% of the test suites. Fault 17 is assigned a suspiciousness value greater than 0.9 for all 100% of the test suites. The final bar in Figure 36 shows the effect of rerunning Tarantula on the program containing fault 14 after removing fault 17. Therefore, for this version, in the 10% of the test suites when only one fault is illuminated by the technique, the illuminated fault can be located and removed, thus allowing the technique to be reapplied to locate the second fault. This phenomenon is an example of the effectiveness of the technique, as at least one of the faults is illuminated for this version. In cases where fewer than all of the faults are revealed by this technique, the user could iteratively remove the discovered faults, retest, and reapply the technique until the test suite passes on all test cases (i.e., sequential debugging) as was described in Section 4.3 on page 64.

To determine how frequently the technique assigns an appropriately low suspiciousness value to non-faulty statements for the program with multiple faults, we examine the suspiciousness values assigned to all non-faulty statements. Figure 35(e)-35(h) displays these results. For all multi-fault versions, we again notice the low number of statements with high suspiciousness values: less than 20% with a suspiciousness value between 0.8 and 1.0. The low percentage of suspicious statements substantially reduces the search space of the program.

Overall, these empirical studies indicate that the technique is effective in illuminating the fault or directing attention toward the fault, and narrowing the search space of the program for the fault.

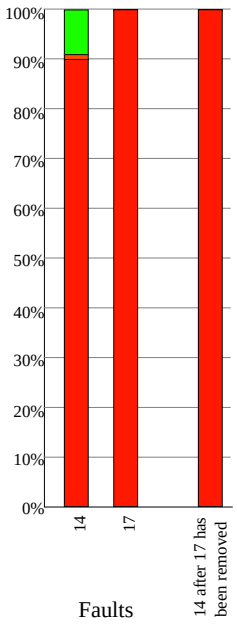

Figure 36: Resulting suspiciousness values for each individual fault in a two-fault version (left); resulting suspiciousness values for the remaining fault after the discovered fault has been removed (right).

7.2.5 Threats to Validity

Threats to external validity arise when the results of the experiment are unable to be generalized to other situations. In this study, we evaluated the effectiveness of the Tarantula technique on one subject program, `space`. The results obtained using the `space` program cannot be generalized to arbitrary programs. Programs of different types, of different sizes, and written in different languages need to be examined with the Tarantula technique to be able to provide some evidence of the generality of the results.

Threats to construct validity arise when the metrics used for evaluation do not accurately capture the concepts that they are meant to evaluate. A threat to construct validity is that we consider the program instructions that differ between the faulty and non-faulty version of the program to be the fault that we are attempting to localize. However, it should be noted that the program could be debugged in a number of ways

to equal effect, including rewriting the entire program. The choice of how to define the fault can affect the results either positively or negatively.

7.3 Study 2: Evaluating the Relative Effectiveness of the Technique Compared to Other Fault-Localization Techniques

To evaluate the relative effectiveness of the Tarantula technique, we compared its effectiveness with other fault-localization techniques. This study compares a number of 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 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 second-most-effective technique studied. At the 90%-score level, Tarantula performs 57% better than the second-most-effective technique studied.

7.3.1 Object of Analysis

For the object of analysis, we used the Siemens suite [37] of programs. We chose these programs because they are the most common object of analysis for comparing fault-localization techniques. Of these faulty Siemens 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 used the remaining 122 versions for our

studies.

7.3.2 Variables and Measures

7.3.2.1 Independent Variables

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

1. Set union (from Reference [61])
2. Set intersection (from Reference [61])
3. Nearest Neighbor Queries (from Reference [61])
4. Cause Transitions (from Reference [17])
5. Tarantula

The Set-union and Set-intersection techniques were described in Section 2.5.1. The Nearest Neighbor Queries technique was described in Section 2.5.2. The Cause Transitions technique was described in Section 2.6. The Tarantula technique was described in Chapter 3.

7.3.2.2 Dependent Variables and Measures

To compare these techniques, we use one dependent variable: effectiveness of the technique in locating the fault. To evaluate the effectiveness of the techniques, we rank the statements of a program in terms of how the individual techniques compute 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 [61] and [17]; we described this ranking technique in Section 2.5.1. 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 Chapter 3. This ranking system uses the "suspiciousness" scores to rank the executable statements in the program.

The effectiveness is determined by a metric presented originally by Renieris and Reiss [61] and used in several other studies (e.g., [17, 44, 49]). The metric, *Score*, is defined as the percentage of the program that need *not* be examined to find the fault using the rank described in the preceding discussion; it is computed by the following equation:

$$Score = \left(1 - \frac{\text{rank of fault}}{\text{size of program}}\right) * 100 \tag{7.3.1}$$

In Equation 7.3.1, *rank of fault* is the placement of the fault in the sorted list of coverage entities from most suspicious to least, and *size of program* is the number of coverage entities in the sorted list.

7.3.3 Experimental Setup

For the Set-union, Set-intersection, and Nearest-Neighbor techniques, we use the results given in Reference [61]. For the Cause-Transitions technique, we use the results given in Reference [17].

For the Tarantula technique, we used the implementation of Tarantula that interfaces with the GNU C compiler instrumenter. 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." We distinguished the statements that are executable and uncovered from statements that are not executable, such as comments, variable declarations, and blank lines. We label only those statements that are executable and uncovered as "uncovered." Each executable statement is given a suspiciousness score and then ranked according to the

ranking system described in Section 3.4. Shell scripts automate running the Tarantula tool on all versions.

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. To demonstrate, consider the example program in Figure 4 on page 27. The Tarantula technique assigns a rank of 1 to the faulty statement out of a total of 13 executable statements. Thus, the score in this case is $(1 - 1/13) * 100 = 92.3\%$.

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.

We identified the faults and failures by using the versions of the subject programs that are deemed to be "correct." To identify the faults, we compared the faulty version of the program with the correct version. The lines in which they differ are recorded as the fault. In the cases where the fault comprised multiple lines, the rank of the fault is defined as the first line to be reached in the sorted list. To distinguish failed from passed 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.

7.3.4 Results and Discussion

Table 3 and Figure 37 show the results concerning the effectiveness dependent variable, Score. Table 3 shows the percentage of versions that achieve a score within each segment listed. Following the convention used by both References [61] and [17], 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 to achieve[1] for all techniques considered, the first segment from 99-100% effectively "pinpoints" the fault in the program.

For example, in Table 3, 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 3 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 37 depict the data in Table 3. 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

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

Table 3: 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

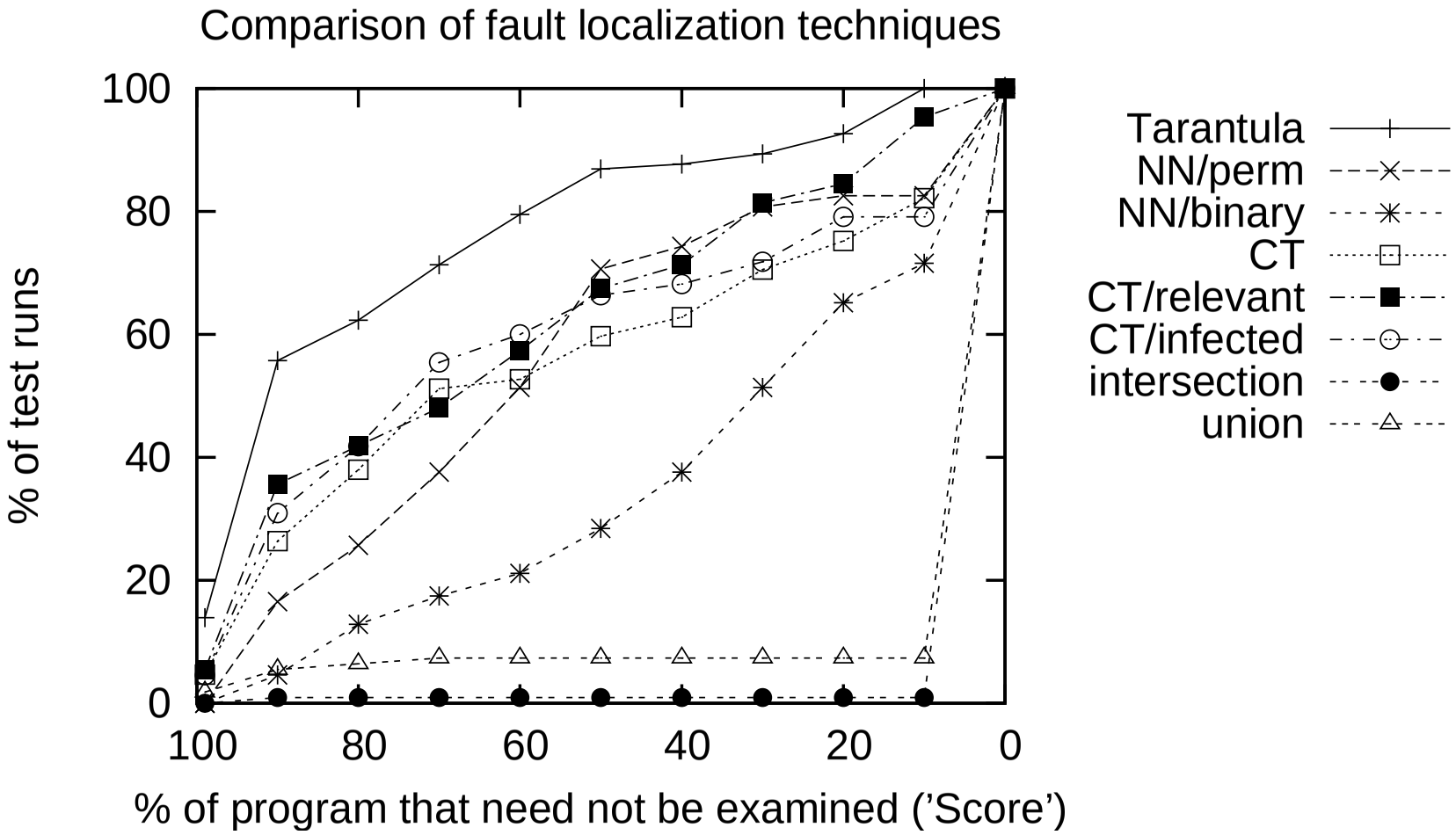

Figure 37: Comparison of the effectiveness of each technique.

the ranking technique.

The horizontal axis represents the score measure defined above, which in turn 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 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 of these techniques is dependent on which single failed test that is used). For these techniques 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 37 shows that the Set-intersection techniques perform the worst, followed by Set-union, Nearest-Neighbor using binary distancing, Nearest-Neighbor using permutation distancing, the Cause-Transitions using different ranking strategies (some that leverage programmer knowledge), and finally, 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.

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 Sections 2.5.1 and 2.5.2. 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. To address this problem, 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. Providing a narrower ordering (closer to a total order) of program regions may better guide the programmer from the regions that are most likely to be faulty to those least likely to be faulty (according to some approximation measures), and may result in better empirical results.

- *Use of single failed test case.* We have found that the Tarantula technique performs better with more failed test cases as well as more passed test cases. With Tarantula, 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. However, we have found that utilizing the information from multiple failed test cases lets the technique leverage the richer information base.

7.3.5 Threats to Validity

There are a number of threats to the validity of this experiment. Specifically, a threat to external validity is that 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 be more

effective. This expectation is supported by the results presented in Section 7.2 and the results summarized in Figure 38. In this figure, the Tarantula technique is applied to the `space` program. 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.

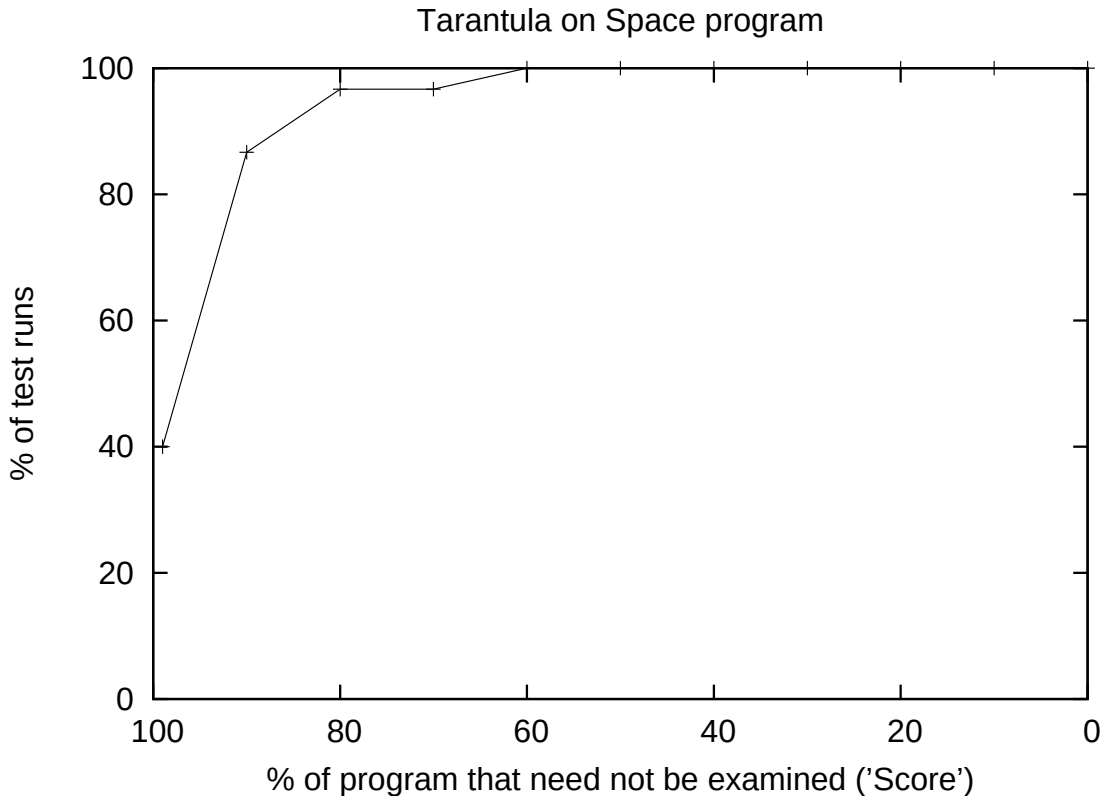

Figure 38: Results of the Tarantula technique on a larger program, `space`.

Another threat to external validity is that 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, the study presented in Section 7.2, 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.

A threat to construct validity is that 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.

7.4 Study 3: Evaluating the Effectiveness of the Clustering Technique for Multiple Faults

To evaluate the effects of clustering failed test cases to create fault-focusing clusters and specialized test suites on fault localization for programs with multiple faults, we compared the effectiveness of the Tarantula technique with and without clustering. We compared three methods of performing the fault localization: (1) with no clustering, (i.e., performing the fault localization using the entire test suite); (2) with the profile-based clustering to create specialized test suites that are used for fault localization; and (3) with the fault-localization-based clustering to create specialized test suites that are used for fault localization. The results of this study show that clustering failures provides cost-saving potential for locating faults in programs with multiple faults.

7.4.1 Object of Analysis

Our experimental protocol created 100 8-fault versions of `space` by choosing from the available faults at random. We simulated a developer's test suite for each version by choosing a test suite at random from a collection of 1000 branch-adequate test suites, each with an average of 156 test cases.

7.4.2 Variables and Measures

7.4.2.1 Independent Variables

Our studies manipulated one independent variable—the technique for creating the fault-focusing clusters that drive the debugging process. The techniques that we examine are

no-cluster : sequential debugging without any clustering

profile-cluster : sequential debugging using profile-based clustering (Section 4.2.1)

fault-cluster : sequential debugging using fault-localization-based clustering (Section 4.2.3)

7.4.2.2 Dependent Variable

In Section 7.3, we defined the *Score* metric that has been used in several empirical studies of the effectiveness of fault-localization techniques (e.g., [17, 44, 49]). In each of these studies and in the study in Section 7.3, the techniques were used to find just one fault. The Score metric is defined as the percentage of the program that need *not* be examined to find the fault using the rank described in Section 3.4. Equation 7.3.1 is used to calculate the Score.

To evaluate the effectiveness of the localization of multiple faults, we use a variation of this metric. Instead of evaluating the fault-localization effectiveness in terms of the percentage of the program that need *not* be examined to find the fault (as described by Equation 7.3.1), we use its complement: the percentage of the program

that *must* be examined to find the fault. This value is indicative of the time or effort that the developer would spend in finding a single fault in the program if she examined the program using the ranks computed by the fault-localization technique. This metric, which we call *Expense*, is computed by the following equation:

$$Expense = \frac{\text{rank of fault}}{\text{size of program}} * 100 \tag{7.4.1}$$

We calculate a metric to assess the *total developer expense* and denote this metric as D. D is used to assess the total of all developers' efforts to find the faults in a program. D is computed as the sum of the developer *Expense* for each fault in the program, and is computed by the following equation:

$$D = \sum_{i=1}^{|faults|} Expense_i \tag{7.4.2}$$

7.4.3 Experimental Setup

We started with an 8-fault version, ran it with the test suite, and detected and removed one fault. Then we generated the 7-fault version using the remaining seven faults and ran it with the same test suite. We repeated the sequential fault-removal process, creating the 6-, 5-, 4-, 3-, 2-, and 1-fault versions until no executions in the test suite failed.

We gathered the D scores for each of the 90 versions and report the mean and standard deviation. We also computed the pair-wise difference of the three different techniques' scores. The $D_{no-cluster}$, $D_{profile-cluster}$, $D_{fault-cluster}$, and their pair-wise differences can be taken as a sample of the entire population of all 8-fault versions for our subject program. Because our sample size is adequately large, their distribution approximates a normal distribution. Thus, we computed a two-sided t-interval with a confidence level of 99%. Interpret this statistic to mean that with 99% confidence, the mean of the sample will be in the range defined by the lower and upper bounds.

For the samples calculated by differencing two data points, if both bounds have the same sign, then we have confidence (with 99% certainty) that one mean is larger than the other for the entire population.

7.4.4 Results and Discussion

Table 4 (and Figure 39) shows the comparative results for total developer expense D. The columns show the sample source, mean, and standard deviation, followed by the lower and upper 99% confidence interval bounds calculated for a two-sided t-interval for the mean of the sample. The first three rows show statistics derived by measuring D for each of the three modes. For example, for the technique with no clustering, the sample mean of $D_{no-cluster}$ for the 90 8-fault versions of `space` is 36.63 with standard deviation of 22.35. The two-sided t-interval with a confidence level of 99% for this mean is between 31.06 and 42.20.

Table 4: Total developer expense, D.

Source	Mean	Std. Dev.	99% lower	99% upper
$D_{no-cluster}$	36.63	22.35	31.06	42.20
$D_{profile-cluster}$	31.50	26.63	24.86	38.14
$D_{fault-cluster}$	26.43	22.42	20.84	32.02
$D_{no-cluster} - D_{profile-cluster}$	5.13	15.49	1.27	8.99
$D_{profile-cluster} - D_{fault-cluster}$	10.20	13.54	6.82	13.57
$D_{fault-cluster} - D_{fault-cluster}$	5.07	13.14	1.80	8.34

The last three rows show statistics about the pair-wise differences among the individual means of the three debugging modes. For example, in the fourth row, the difference between the means $D_{profile-cluster}$ and $D_{no-cluster}$ is 5.13, with a standard deviation of 15.49. The two-sided t-interval for the difference of the means $D_{no-cluster} - D_{profile-cluster}$ is between 1.27 and 8.99

The results show that the most expensive technique is the *no-cluster* mode and the least is the *fault-cluster* mode. When comparing the means of these two debugging

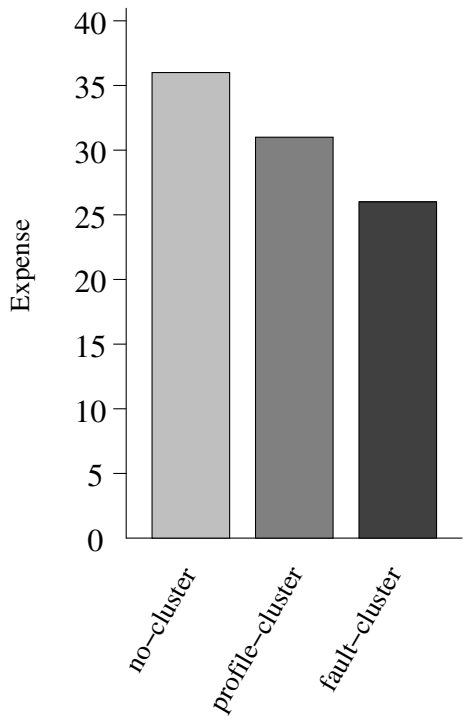

Figure 39: Mean score for the total developer expense, D, for the three techniques.

techniques in row five, we see that $D_{no-cluster}$ is expected, with a 99% confidence, to be greater than $D_{fault-cluster}$ by a value between 6.82 and 13.57. These results mean that the use of fault-focusing clusters and the resultant test suites yields reduced total developer expense.

7.4.5 Threats to Validity

Although this empirical study provides evidence of the potential usefulness of the execution clustering techniques developed in this research, there are several threats to the validity of the empirical results that should be considered in their interpretation.

Threats to the external validity of an experiment limit generalizing from the results. The primary threat to external validity for this study arise because only one medium-sized C program has been considered. Thus, we cannot claim that these results generalize to other programs. In particular, no generalization can be made as to the effectiveness of clustering for debugging. However, a variety of faults were

randomly combined to produce the 100 8-fault versions used in this research, and thus, these versions are useful for exploring the presented techniques.

Also, we assume that a developer can identify the fault by inspecting the code—that is, he can follow the order of statements that is specified and determine at each one whether it is faulty. We think that the amount of code that must be examined while following the prescribed order of the fault-localization technique is indicative of the technique's effectiveness, and other researchers use the same methodology to evaluate fault-localization techniques (e.g., [17, 49, 61]).

7.5 Study 4: Evaluating the Effectiveness of the Clustering Technique for Debugging in Parallel

To evaluate the potential to parallelize the debugging process, we compared the effectiveness of debugging with the Tarantula technique in the sequential mode and in the parallel mode of debugging. We compared three modes of debugging: (1) debugging in sequence with no clustering; (2) debugging in parallel utilizing profile-based clustering; and (3) debugging in parallel utilizing fault-localization-based clustering. This study finds that clustering failures for the purpose of parallelizing the debugging process can provide a significant savings by limiting the cost of locating faults in programs with multiple faults.

7.5.1 Object of Analysis

We use the same 100, randomly generated, 8-fault versions of `space` as those used for Study 3, described in Section 7.4.1.

7.5.2 Variables and Measures

This study manipulated one independent variable—the technique for creating the fault-focusing clusters that drives the debugging process. The techniques that we examined are

sequential : sequential debugging without any clustering

profile-parallel : parallel debugging using profile-based clustering (Section 4.2.1)

fault-parallel : parallel debugging using fault-localization-based clustering (Section 4.2.3)

To evaluate the effectiveness of the localization of multiple faults in parallel, we use the *Expense* measure that was defined in Section 7.4.2. Using the Expense measure, we compute another metric to assess the *critical expense to a failure-free program* and denote this metric as FF. FF is used to assess the relative savings in terms of the time to deliver a failure-free program (i.e., the expense of the critical path in a failure-free program). FF is computed as the sum of the maximum developer expense at each debugging iteration,[2] which is the critical path to achieving a failure-free program, and is computed by the following equation:

$$FF = \sum_{i=1}^{|iterations|} \max\{Expense_f | f \text{ is a fault subtask at iteration } i\} \tag{7.5.1}$$

We also use the total developer expense measure D that was defined in Section 7.4.2 to evaluate these different modes of debugging.

Note that, when debugging without any clustering (using the whole test suite), the D and FF values are always equal—the total developer expense and the critical expense of a failure-free program are equal because both are calculated as the sum of the one-at-a-time developer expenses.

These two metrics, total developer expense (D) and critical expense to a failure-free program (FF), capture the two important dimensions of debugging in parallel. In this study, the sequential mode of debugging does not use any clustering. Thus, $D_{sequential}$ and $FF_{sequential}$ will necessarily be equal.

[2]Section 4.4 describes the phenomenon of fault dominance, and Figure 22 (on page 69) shows an example of why more than one debugging iteration may be necessary.

It is worth noting that the *Expense* metric also accounts for errors in the clustering process. Particularly, when the clustering approach produces multiple clusters for finding the same fault, multiple developers would be simultaneously and unknowingly debugging the same fault. Because each such developer is working independently, their expense is not as efficient as if only one of them were debugging that fault. When the first of these redundant efforts results in a found fault, the other developers that are also working on that fault had expended redundant effort. We assume that the developers that are simultaneously debugging communicate with one another when a fault has been found. This communication limits the expense of any other developer that may be working to find the same fault. When a developer receives notice from another developer that a fault has been found, he can check to see if that fault is the one causing his failures, and if so, stop his debugging efforts. Thus, the expense metric is calculated as the product of the minimum of the redundant effort and the number of developers working on that fault. Figure 40 shows an illustration of three clusters that target the same fault and a fourth cluster that targets another fault. In this example, the expense required to find fault 1 is calculated as $Expense_1 = 3 * \min(2, 3, 4) = 6$, where the minimum of these three developers' expenses is depicted with the dotted line at a value of 2. The total developer expense for this example is calculated as $D = Expense_1 + Expense_2 = 6 + 5 = 11$. Thus, we capture the inherent inefficiencies that sometimes occur because of inaccurate clustering.

7.5.3 Experimental Setup

Like the experimental setup presented for Study 3, our experimental protocol created 100 8-fault versions of `space` by choosing from the available faults at random. We simulated a developer's test suite for each version by choosing a test suite at random from a collection of 1000 branch-adequate test suites, each with an average of 156 test cases. Over the course of evaluating all debugging modes, we generated 1147

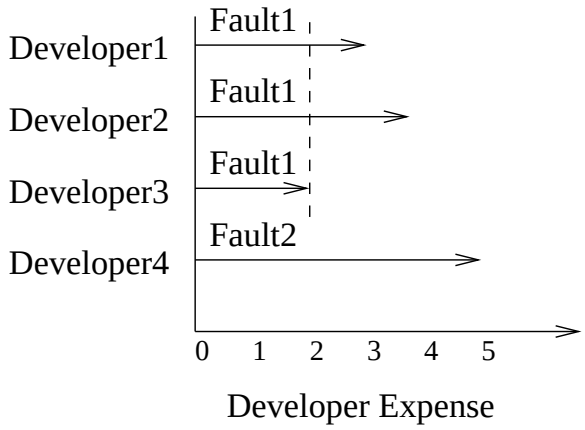

Figure 40: The cost model accounts for when the clustering technique produces multiple test suites that target the same fault.

derivative multi-fault versions. For example, in the sequential mode, we started with an 8-fault version, ran it with the test suite, and detected and removed one fault. Then we generated the 7-fault version using the remaining seven faults and ran it with the same test suite. The sequential fault-removal process repeated, creating the 6-, 5-, 4-, 3-, 2-, and 1-fault versions until no executions in the test suite failed. In the two parallel modes, we also started with the 8-fault version, determined the number of clusters and found the faults that they focused, removed those faults, and repeated the process with another iteration of debugging in parallel with a derivative faulty version containing only the remaining faults.

To determine the best threshold parameterization, as described in Section 3.1, we sampled various parameters using ten 8-fault versions. We selected from these the best candidates for use in our study of the remaining 90 8-fault versions. This "training" of the parameters for a program is similar to the way in which we would prescribe training in the field. For both clustering techniques, *profile-based clustering* and *fault-localization-based clustering*, we used only the top 20% of the most suspicious lines—$MostSusp = 20\%$. For determining the stopping criterion for the clustering, we used a threshold of $Sim = 68\%$ (roughly two standard deviations) in the *Jaccard* similarity scores. Also informed by the training, for clustering based on sets of suspicious code,

we used a threshold of $Sim = 50\%$ in the *Jaccard* similarity scores.

We gathered the D and FF scores for each of the 90 versions and report their means and standard deviations. We also computed the pair-wise differences of the three different techniques' scores. The $D_{sequential}$, $D_{profile-based}$, $D_{fault-based}$, and their pair-wise differences (and likewise for FF) can be considered as a sample of the entire population of all 8-fault versions for our subject program. Because our sample size is adequately large, their distribution approximates a normal distribution. Thus, we computed a two-sided t-interval with a confidence level of 99%. Interpret this statistic to mean that with 99% confidence, the mean of the sample will be in the range defined by the lower and upper bounds. For the samples calculated by differencing two data points, if both bounds have the same sign, then we have confidence (with 99% certainty) that one mean is larger than the other for the entire population.

7.5.4 Results and Discussion

Our two principal metrics for comparing the costs of the three investigated modes are total developer expense D and critical expense to failure-free FF.

7.5.4.1 Total developer expense

Note that because the total developer expense is necessarily the same for sequential debugging and parallel debugging, the results presented here mirror those that were presented in Section 7.4.4. They are presented here for convenience and because they have relevance in the context of evaluating the sequential versus parallel modes of debugging.

Table 5 (and Figure 41) shows the comparative results for total developer expense D. The columns show the sample source, mean, and standard deviation, followed by the lower and upper 99% confidence interval bounds calculated for a two-sided t-interval for the mean of the sample. The first three rows show statistics derived by measuring D for each of the three modes. For example, for the sequential debugging

Table 5: Total developer expense, D.

Source	Mean	Std. Dev.	99% lower	99% upper
$D_{sequential}$	36.63	22.35	31.06	42.20
$D_{profile-parallel}$	31.50	26.63	24.86	38.14
$D_{fault-parallel}$	26.43	22.42	20.84	32.02
$D_{sequential} - D_{profile-parallel}$	5.13	15.49	1.27	8.99
$D_{sequential} - D_{fault-parallel}$	10.20	13.54	6.82	13.57
$D_{profile-parallel} - D_{fault-parallel}$	5.07	13.14	1.80	8.34

mode, the sample mean of $D_{sequential}$ for the 90 8-fault versions of `space` is 36.63 with standard deviation of 22.35. The two-sided t-interval with a confidence level of 99% for this mean is between 31.06 and 42.20.

The last three rows show statistics about the pair-wise differences among the individual means of the three debugging modes. For example, in the fourth row, the difference between the means $D_{profile-parallel}$ and $D_{sequential}$ is 5.13, with a standard deviation of 15.49. The two-sided t-interval for the difference of the means $D_{sequential} - D_{profile-parallel}$ is between 1.27 and 8.99

The results show that the greatest developer expense is with the *sequential* mode and the least is with the *fault-parallel* mode. When comparing the means of these two debugging modes shown in row five, we see that $D_{sequential}$ is expected, with a 99% confidence, to be greater than $D_{fault-parallel}$ by a value between 6.82 and 13.57. These results mean that the use of fault-focusing clusters and the resultant test suites yields reduced total developer expense even if the debugging is done by a single developer.

7.5.4.2 Critical expense to a failure-free program

Table 6 (and Figure 41) presents the comparative results for the critical expense to a failure-free program *FF*. The table is constructed identically to Table 5. Here, for example, for the sequential debugging mode, the sample mean for $FF_{sequential}$ for

the 90 8-fault versions of `space` is 36.63 with standard deviation of 22.35. The two-sided t-interval with a confidence level of 99% for this mean is between 31.06 and 42.20. Note that $D_{sequential}$ and $FF_{sequential}$ are necessarily identical, as explained in Section 7.5.2 (on page 125).

The results show that the greatest critical expense is with the *sequential* mode and the least is with the *profile-parallel* mode. When comparing the means of these two debugging modes shown in row four, we see that the mean of $FF_{sequential}$ is expected, with a 99% confidence, to be greater than the mean of $FF_{profile-parallel}$ by a value between 14.96 and 21.92. Furthermore, when we compare the means of the two parallel debugging modes shown in row six, we see that $FF_{profile-parallel}$ is expected, with a 99% confidence, to be *less* than $FF_{fault-parallel}$ by a value between 0.84 and 4.69 (negating the values shown.) These results mean that both parallel modes are better than the sequential mode, and that for this subject *profile-parallel* outperforms *fault-parallel* in terms of FF. The results show that the use of the *profile-parallel* debugging mode yields a 50% reduction in the critical expense to a failure-free program over the *sequential* mode.

Table 6: Critical expense to failure-free, FF.

Source	Mean	Std. Dev.	99% lower	99% upper
$FF_{sequential}$	36.63	22.35	31.06	42.20
$FF_{profile-parallel}$	18.19	13.74	14.76	21.61
$FF_{fault-parallel}$	20.95	15.00	17.21	24.69
$FF_{sequential} - FF_{profile-parallel}$	18.44	13.96	14.96	21.92
$FF_{sequential} - FF_{fault-parallel}$	15.68	12.03	12.68	18.68
$FF_{profile-parallel} - FF_{fault-parallel}$	-2.76	7.72	-4.69	-0.84

Notable in the results is that the *sequential* mode is the most expensive both in the developer expense, D, and in the critical expense to failure-free, FF. Both parallel techniques provide a savings over the sequential mode. This means that the fault-focusing ability of either clustering technique has economic benefits as measured in

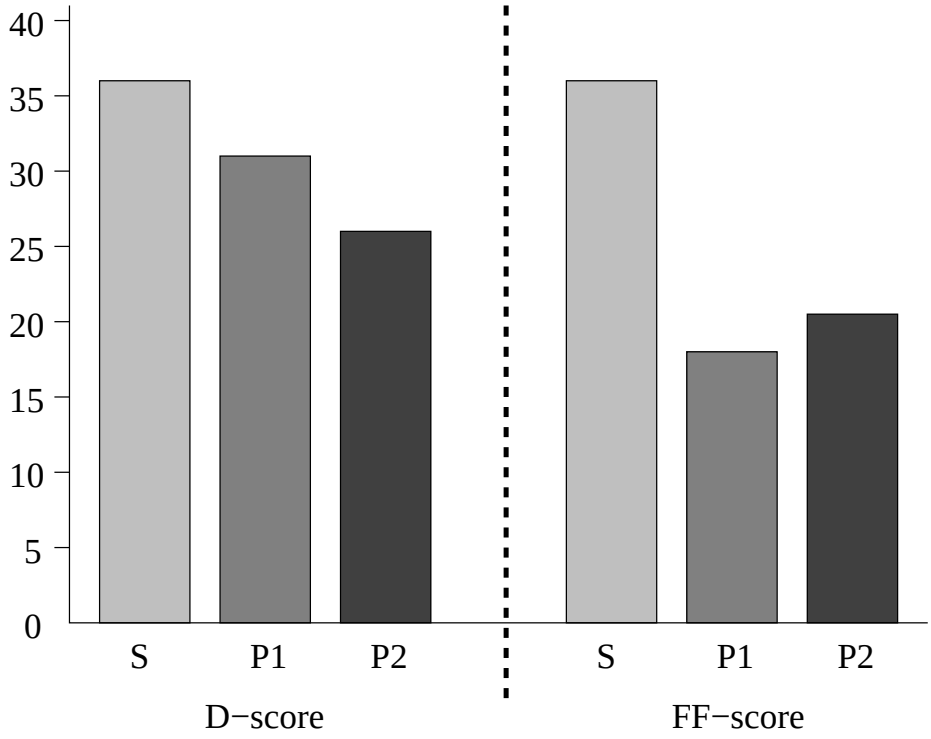

Figure 41: Mean score for the total developer expense, D, and the critical expense to failure-free, FF, for the three techniques.

expense.

The choice of *profile-parallel* or *fault-parallel* may depend on the development organization's resources and circumstance. If the goal is to deliver a failure-free program as fast a possible, then *profile-parallel* may be a better choice than *fault-parallel*. However, if the goal is to minimize development expense, then *fault-parallel* may provide a net savings. We investigated this trade-off to determine the reason that each technique demonstrated a different strength. The *fault-parallel* technique seems to cluster more aggressively than *profile-parallel*. *profile-parallel* and *fault-parallel* respectively have an average of 2.08 and 1.62 parallel fault subtasks across all versions and iterations. *profile-parallel* may incur more total expense due to the under-clustering situation described in Section 7.5.2 and depicted in Figure 40. Moreover, because each of the techniques that we implemented has merit, it is likely that clustering executions for the purpose of fault-localization may be conducted in a number of ways

with good results. Although more research is necessary to determine the best clustering technique, we have demonstrated the promise of parallelizing the debugging effort in such an automated way.

7.5.5 Threats to Validity

Although this empirical study provides evidence of the potential usefulness of the parallel-debugging techniques developed in this research, there are several threats to the validity of the empirical results that should be considered in their interpretation.

Threats to the external validity of an experiment limit generalizing from the results. The primary threat to external validity arises because only one medium-sized C program has been considered. Thus, we cannot claim that these results generalize to other programs. In particular, no generalization can be made as to the effectiveness of parallel debugging. However, a number of faults were randomly chosen from the subject and combined to produce the 100 8-fault versions used in this research and thus, these versions are useful for exploring the presented techniques.

Threats to the internal validity occur when there are unknown causal relationships between independent and dependent variables. In this study, we have postulated a simplistic development scenario that removes these causal relationships. However, for real developers, there will be causal relationships between total expense and the debugging mode chosen. For example, developers will interact with each other, which may change the expense in either direction.

Also, we assume that a developer can identify the fault by inspecting the code—that is, she can follow the order of statements that is specified and determine at each one whether it is faulty. We do think that the amount of code that must be examined while following the prescribed order of the fault-localization technique is indicative of the technique's effectiveness. This issue should be explored further with human studies.

The integration of multiple bug fixes may be more error-prone than one-at-a-time bug fixing. This may cause new bugs to be introduced as the parallel debugging proceeds. Our experiment does not address this difficulty; further studies are needed to explore it.

7.6 Study 5: Evaluating the Efficiency of the Tarantula Technique

To evaluate the efficiency of the Tarantula technique, we recorded timings of applying the technique to subject programs and test suites. We compare these results with the published timings of another state-of-the-art fault-localization technique. We found that Tarantula is in fact efficient—in some cases two orders of magnitude more efficient than the compared technique.

7.6.1 Object of Analysis

For the object of analysis, we used the Siemens suite [37] of programs. Of these faulty Siemens versions, we were able to use 122 versions. These versions are described in Section 7.3.1.

7.6.2 Variables and Measures

Our study manipulated one independent variable: the fault-localization technique. The techniques that we examine are: Tarantula and Cause Transitions [17]. We also reason about the timings of three other techniques: Set union, Set intersection, and Nearest Neighbor Queries.

To compare these techniques, we use one dependent variable: the efficiency of the technique, measured as units of time to perform the analysis.

7.6.3 Experimental Setup

To evaluate the efficiency of the techniques, we recorded timings of using the Tarantula technique. The timings are gathered for both computational time and time required

for necessary I/O. For the Cause-Transitions technique, we use the timing averages reported in [17]. 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 7.6.4.

7.6.4 Results and Discussion

Table 7 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.

Table 7: 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

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 their timings would be 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 passed test cases and a single failed test case. In the Nearest-Neighbor technique, a distance score must be defined for every passed 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 passed test cases to determine which passed test case will be chosen as the "nearest" one to the failed test case used. Similarly for the Set-union and Set-intersection techniques, coverage information for all passed test cases and one failed 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.

7.6.5 Threats to Validity

A limitation to this study is that we did not implement the technique to which we compared. 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 7) that we believe that the implementation cannot explain these differences.

7.7 Study 6: Evaluating the Efficiency of the Clustering Techniques for Multiple Faults and Parallel Debugging

To evaluate the efficiency of the two clustering techniques we measured the timings of applying the clustering techniques. These clustering techniques were defined to improve the effectiveness of fault localization in the presence of multiple faults and to enable the parallelization of the debugging process. We found that the fault-localization-based clustering is more efficient than the profile-based clustering. Moreover, we found that fault-localization-based clustering is efficient enough to perform in practice, requiring, on average, less than a second to compute the clusters.

7.7.1 Object of Analysis

We use the same 100, randomly generated, 8-fault versions of `space` as those used for Study 3, described in Section 7.4.1.

7.7.2 Variables and Measures

This study manipulated one independent variable: the clustering technique used to cluster failed test cases to create fault-focusing clusters. The techniques that we examine are

- profile-based clustering
- fault-localization-based clustering

To compare these clustering techniques, we use one dependent variable: the efficiency of the technique measured by the time required to perform the clustering.

7.7.3 Experimental Setup

For the fault-localization-based clustering, we started with 100 8-fault versions of the subject program, and performed the full parallel-mode debugging: clustering the failures, creating specialized test suites, finding the faults that each of them found,

removing those faults from the program, and then iterating the process until the program and test suite were failure-free. We measured the time for each of the clusterings.

For the profile-based clustering, the implementation of the tool that performed this clustering was no longer available to re-run the experiment and gather the timings. Because of its unavailability, we implemented the clustering technique and simulated the experiment. We simulated its use on the subject by randomly generating 100 branch profiles for the `space` program. These were clustered using the profile-based technique, and the time for this clustering was recorded. This process was repeated 1000 times so that we could take the mean of the timings.

7.7.4 Results and Discussion

Figure 42 summarizes the efficiency results for the study. For the profile-based clustering, the mean time was 24.8 seconds to perform each clustering of 100 failed test cases. For the fault-localization-based clustering, the mean time was 0.8 seconds to perform each clustering of all failed test cases over the entire experiment.

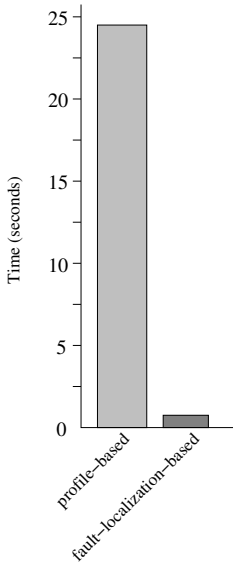

Figure 42: Mean time for clustering expressed in seconds.

These results show that clustering can be performed efficiently for the purposes of debugging programs with multiple faults or for debugging in parallel. Although the profile-based clustering is less efficient than the fault-localization-based clustering, there are likely many other clustering techniques that can be used for this purpose that are more efficient. The fault-localization-based clustering has been found to be both efficient and effective (see Sections 7.4 and 7.5).

7.7.5 Threats to Validity

A limitation of this study is that we were unable to re-run the experiment with the profile-based clustering to gather timings. This could be a factor when considering the efficiency results. However, the simulated environment was very similar to the actual environment for the experiment. In fact, although we did not gather formal timings when the experiment was originally run, the actual profile-based clustering took much longer than our re-implemented version.

7.8 Study 7: Studying the Effects of the Composition of the Test Suite on Fault Localization

To evaluate the effects of the composition of the test suite on fault localization, we performed an 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 this study, we used 10 test-suite reduction strategies and four existing fault-localization techniques including the Tarantula approach, along with a set of programs, containing single and multiple faults, and a large number of test suites. This study shows the trade-offs that exist between test-suite reduction and fault-localization effectiveness. This study also shows that, in general, existing test-suite reduction strategies reduce the effectiveness of fault-localization techniques.

7.8.1 Object of Analysis

We used the Siemens suite of programs and the `space` program. 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.

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.

7.8.2 Variables and Measures

For this study, we manipulated two independent variables: the test-suite reduction strategies, and the fault-localization technique. We examined 10 test-suite reduction strategies. We also examined the effects of reduction on four fault-localization techniques.

7.8.2.1 Independent Variable: 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., [36]), 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 43. This example program is the same as that shown in Figure 4 from Chapter 2 except that this example has a test suite that contains more test cases and shows the fault-localization results from multiple techniques. 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 7.8.2.2. 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

		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() {	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	7,5,4	2,1,3	4,3,5									
	int x,y,z,m;																	
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 43: Example program, information about its test suite, and its rank results for the four fault-localization techniques.

executed by one test case.[3] To illustrate, consider again the program and test suite shown in Figure 43. 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 $\langle s1, s2, s3, s4, s6, s7, s13 \rangle$. 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 program, 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 strategies, 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

[3]Another term for the set of statements executed by a test case is an *execution slice*.

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 43. Table 8 shows, for each test-suite reduction strategy, one possible reduction result.

7.8.2.2 Independent Variable: Fault-Localization Techniques

We examined how the test-suite reduction strategies would effect the ability of four fault-localization techniques to perform effectively. We considered four fault localization techniques:

- Tarantula

Table 8: 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}

- Statistical Bug Isolation
- Jaccard
- Ochiai

The Tarantula technique was defined in Chapter 3. The Statistical Bug Isolation technique, or SBI, was defined in Section 3.6.1. The Jaccard and Ochiai techniques were proposed by Abreu and colleagues for the purposes of fault localization [1] and were defined in Section 3.5.3.

For convenience, the metrics that define the ranking of the coverage entities and thus influence the effectiveness are shown here:

For Tarantula, we define two metrics: suspiciousness and confidence as such:

$$suspiciousness_T(s) = \frac{\%\text{failed}(s)}{\%\text{failed}(s) + \%\text{passed}(s)} \tag{7.8.1}$$

$$confidence(s) = max(\%\text{failed}(s), \%\text{passed}(s)) \tag{7.8.2}$$

For Statistical Bug Isolation, to facilitate comparison among Tarantula, SBI, and the other fault-localization techniques, we adapted Equation 3.6.1 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 $suspiciousness_S$.

$$suspiciousness_S(s) = \frac{failed(s)}{passed(s) + failed(s)} \tag{7.8.3}$$

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.

The Jaccard technique defines suspiciousness of a coverage entity as

$$suspiciousness_J(s) = \frac{failed(s)}{totalfailed + passed(s)} \tag{7.8.4}$$

The Ochiai technique defines suspiciousness of a coverage entity as

$$suspiciousness_O(s) = \frac{failed(s)}{\sqrt{totalfailed * (failed(s) + passed(s))}} \tag{7.8.5}$$

Figure 43 shows the way that each of these techniques assigns suspiciousness to each coverage entity, in this case statements. It also shows for every technique, the ranking that results.

7.8.2.3 Dependent Variables

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.

$$Reduction = \left(1 - \frac{\text{size of reduced test suite}}{\text{size of unreduced test suite}}\right) * 100 \tag{7.8.6}$$

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*, was defined in Section 7.4.2, and is computed by the following equation.

$$Expense = \frac{\text{rank of fault}}{\text{number of executable lines of code}} * 100 \tag{7.8.7}$$

7.8.3 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. The smaller test suites are subsumed by the larger test suites—the 100-test-case test suite contains the 50-test-case test suite, the 150-test-case test suite contains the 100-test case test suite, and so on. 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 7.8.2.1 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.

7.8.3.1 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). We repeated the test case selection, without replacement, 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.

7.8.3.2 Applying Reduction Strategies

For each of the $169,000$ unreduced test suites, we used the following process[4] 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.

[4]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.

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[5] 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.

7.8.4 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 7.8.4.1 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 actually differ. Section 7.8.4.2

[5]This is equivalent to randomly selecting one test case because the test suites we used were randomly generated from Section 7.8.3.1.

Table 9: 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

presents these test-suite reduction results. Based on the results of Sections 7.8.4.1 and 7.8.4.2, we chose two reduction strategies that are representative of the others, and present their effects on each of the four fault-localization techniques. Section 7.8.4.3 presents these results. Finally, in Section 7.8.4.4, we show the size reduction of the test suites by the two representative reduction strategies for each of the subject programs.

7.8.4.1 Expense Increase on Tarantula

Table 9 shows the increase in *Expense* of using the Tarantula 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 7.8.7) 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.

7.8.4.2 Percentage Reduction

Table 10 shows the percentage reduction in the size of the test suite using each of the 10 test-suite reduction strategies. Like Table 9, 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% is achieved on the unreduced test suite by test-suite reduction strategy SP; this means that the reduced test suite is only 7.7% 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 provide about a 50% reduction in the test-suite size. One exception occurs for both statement-based and vector-based reduction when they

Table 10: 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

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.

7.8.4.3 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 7.8.2.2, 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). Tables 9 and 10 indicate that applying these two test-suite reduction strategies on all test cases is representative of the other eight test-suite reduction strategies.

Figures 44, 45, 46, 47, 48, 49, 50, 51, 52, and 53 show these results. We present the data using 10 boxplot[6] charts. Figure 44 through 51 show the charts for each of the single-fault programs, and Figures 52 and 53 show the charts for the multiple-fault versions of the `space` program. 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.

These data show 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

[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.

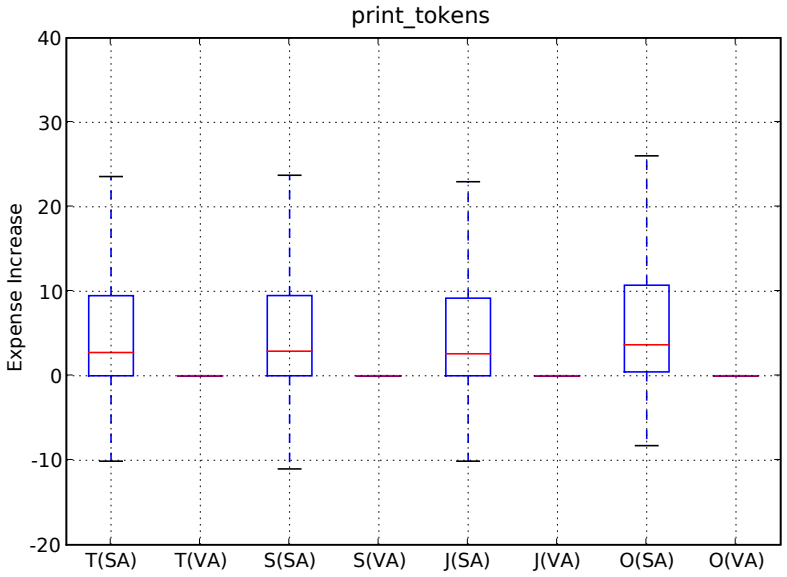

Figure 44: Expense increase for the statement-based reduction (SA) and the vector-based reduction (VA) for single-fault programs for print_tokens.

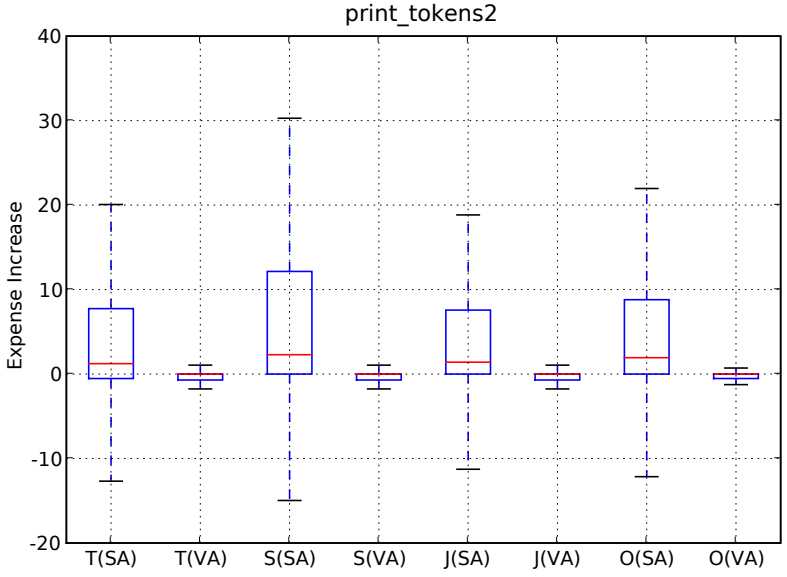

Figure 45: Expense increase for the statement-based reduction (SA) and the vector-based reduction (VA) for single-fault programs for print_tokens2.

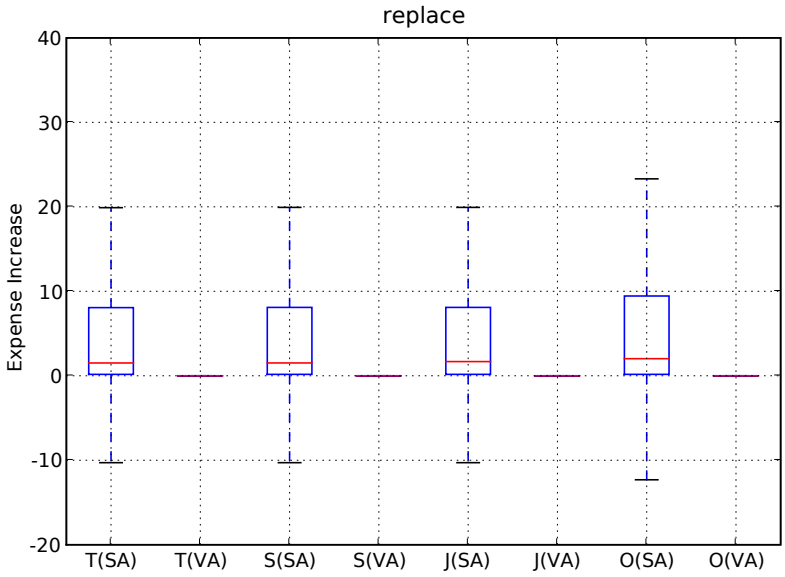

Figure 46: Expense increase for the statement-based reduction (SA) and the vector-based reduction (VA) for single-fault programs for replace.

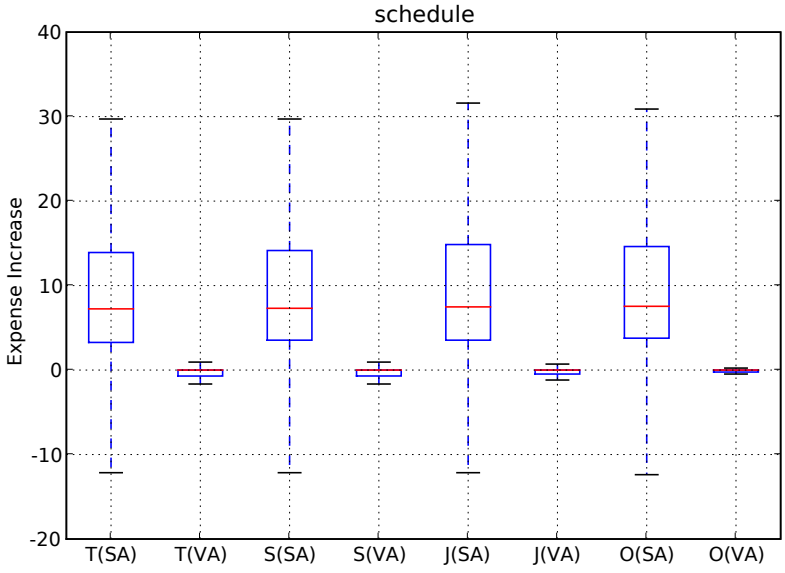

Figure 47: Expense increase for the statement-based reduction (SA) and the vector-based reduction (VA) for single-fault programs for schedule.

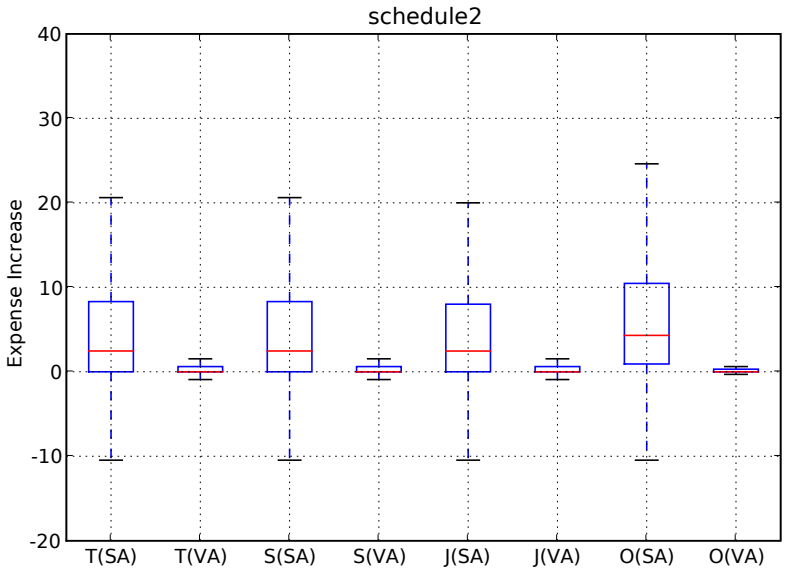

Figure 48: Expense increase for the statement-based reduction (SA) and the vector-based reduction (VA) for single-fault programs for schedule2.

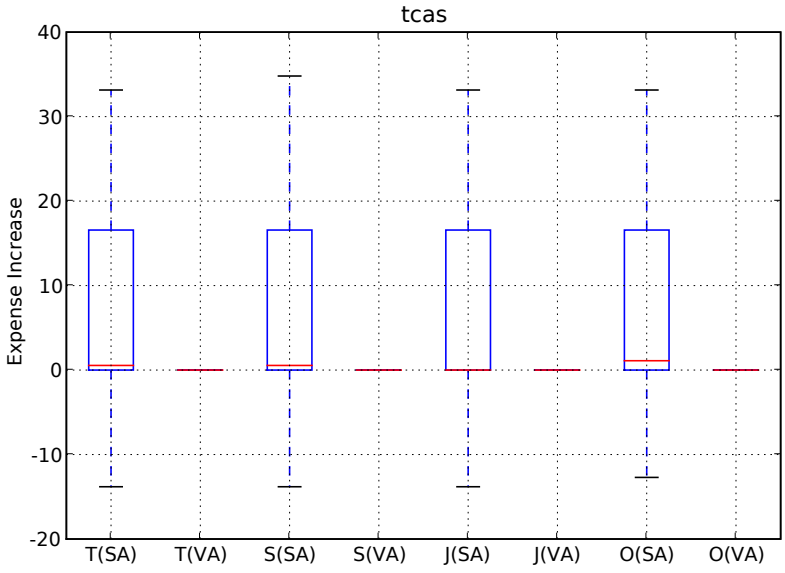

Figure 49: Expense increase for the statement-based reduction (SA) and the vector-based reduction (VA) for single-fault programs for tcas.

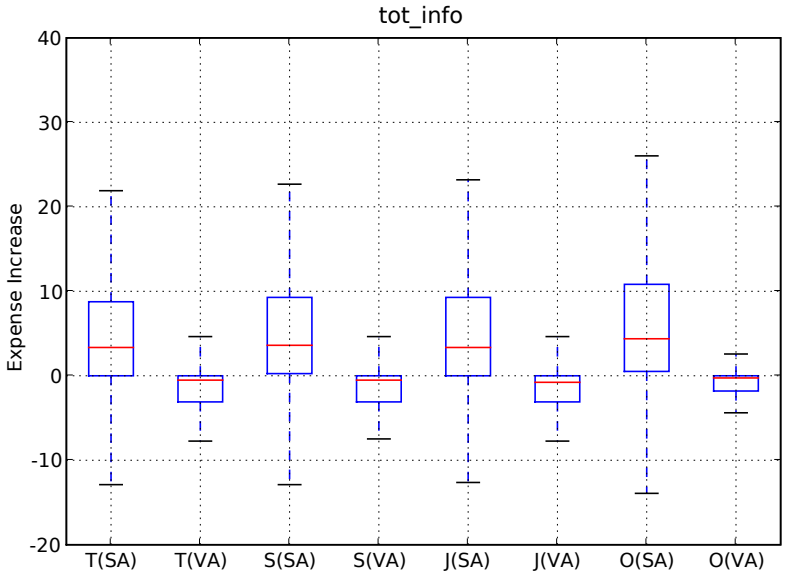

Figure 50: Expense increase for the statement-based reduction (SA) and the vector-based reduction (VA) for single-fault programs for tot_info.

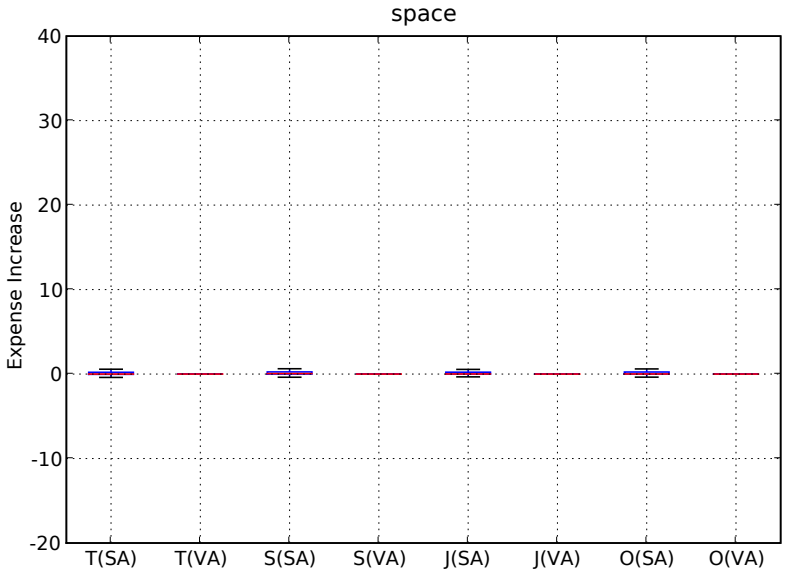

Figure 51: Expense increase for the statement-based reduction (SA) and the vector-based reduction (VA) for single-fault programs for Space.

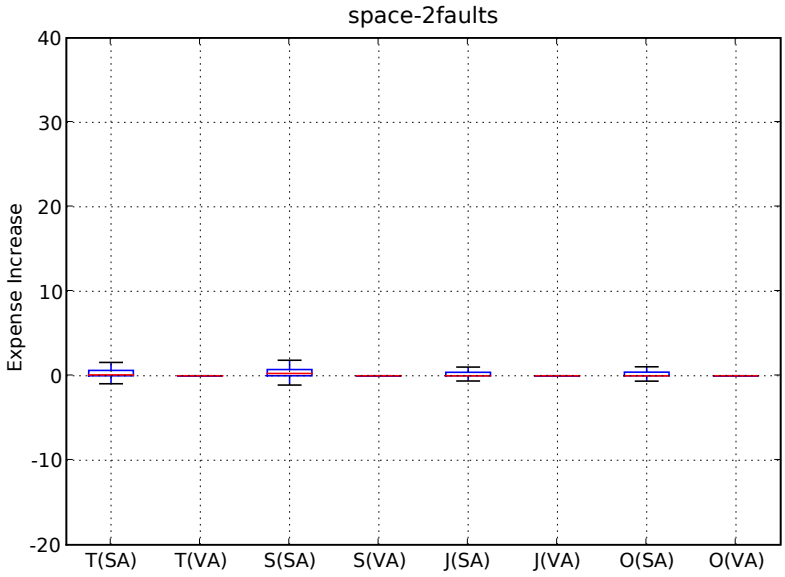

Figure 52: Expense increase for the statement-based reduction (SA) and vector-based reduction (VA) for Space with 2 faults.

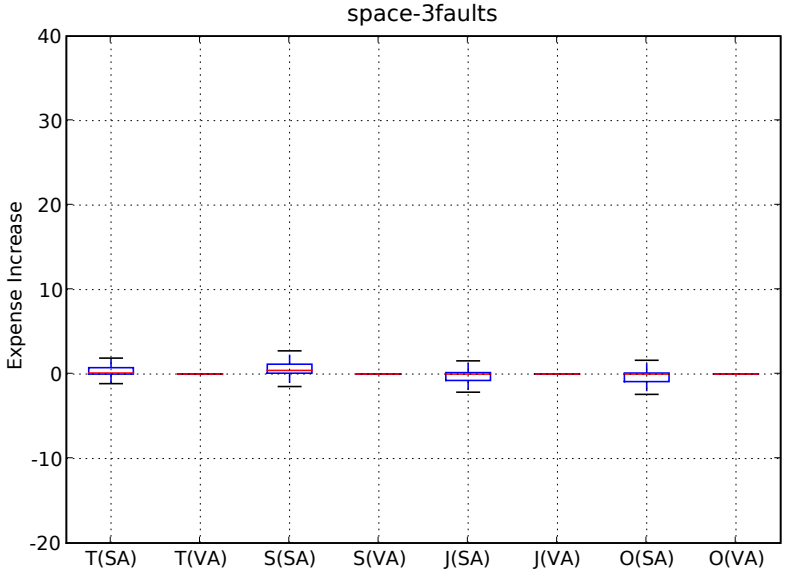

Figure 53: Expense increase for the statement-based reduction (SA) and vector-based reduction (VA) for Space with 3 faults.

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.

7.8.4.4 Size Results

Figure 54 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 reduction strategy provides a much greater and more consistent reduction than the vector-based reduction strategy.

7.8.5 Discussion

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

The data demonstrates 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 [35] found that test-case redundancy can negatively affect fault localization. Our studies provide a more thorough experiment with the goal 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

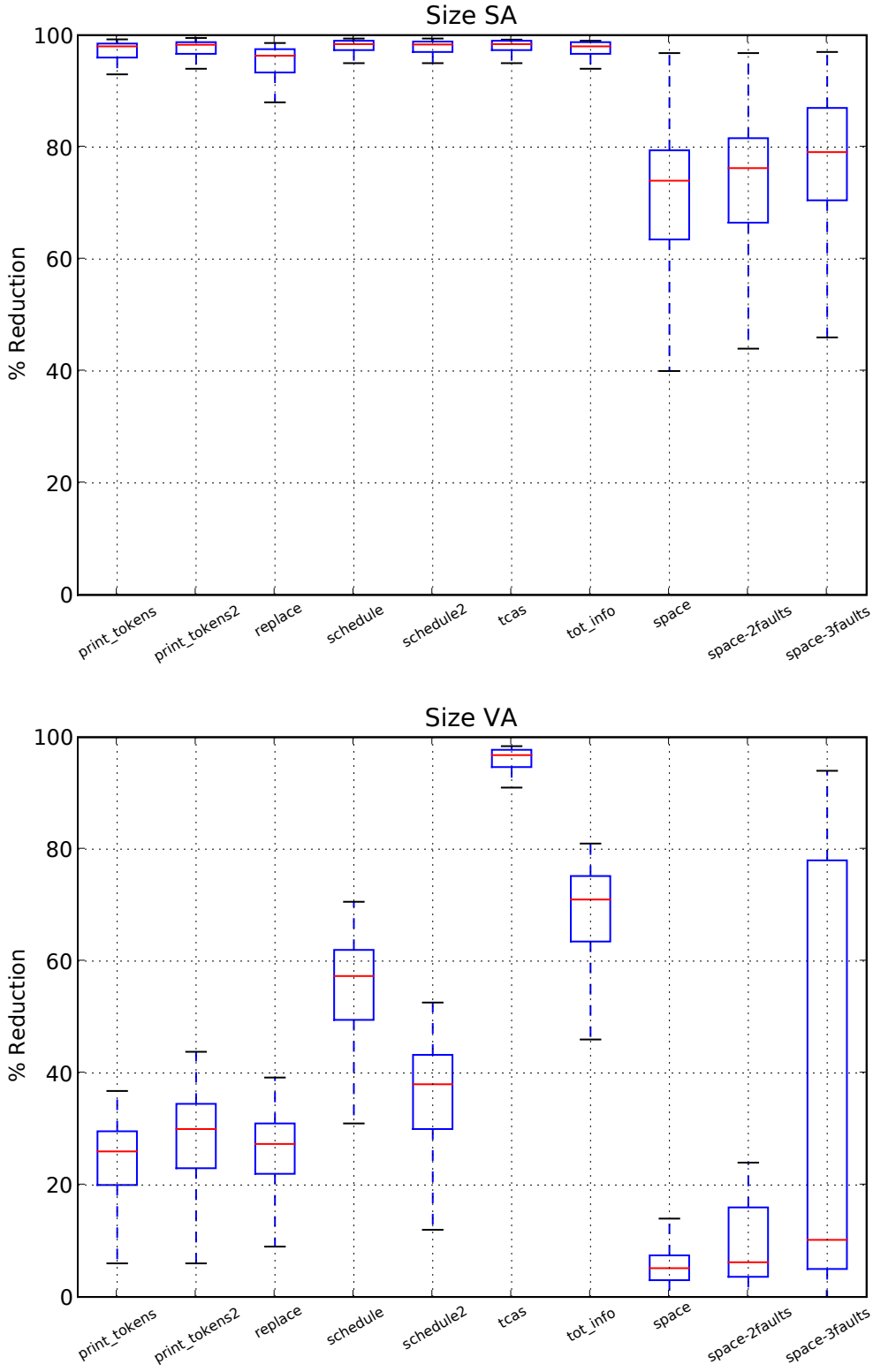

Figure 54: Percentage of test-suite size reduction for statement-based reduction (SA) and vector-based reduction (VA).

removing redundancy, in our study, the improvement was 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.

7.8.6 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.

CHAPTER VIII

CONCLUSIONS

This dissertation presents an approach for providing partial automation for fault localization with the use of commonly available dynamic information gathered from test-case executions in a way that is effective, efficient, tolerant of test cases that pass but also execute the fault, and scalable to large programs that potentially contain multiple faults. Specifically, the approach and techniques identify suspicious regions in programs using coverage information gathered from test cases. These suspicious regions can be reported to the developer to direct attention in the debugging process. This dissertation also provides techniques to automatically partition test suites to better enable fault localization of programs with multiple faults and to enable a parallelized approach to debugging.

These approaches have been shown to be effective and efficient for the programs that we studied. Specifically, the fault-localization technique has been shown to more effective in pinpointing faults than the best of the techniques to which we compared it for the subject programs used. Moreover, in some cases, our technique was found to be two orders of magnitude more efficient than this best technique to which we compared it. The test suite partitioning and the creation of specialized test suites was shown to be an effective and efficient approach to addressing the problem of programs with multiple faults. Our studies showed that clustering failures can provide an expected gain in efficiency over not clustering with a 99% confidence for the subject programs that we used. Moreover, using failure clustering to enable debugging in parallel gives an additional benefit in terms of a reduced critical expense or a reduced time to delivery. Our studies showed a 50% reduction in the critical expense to a failure-free

program for our subject programs. Our studies also found that failure clustering can be efficient: on average the clustering required less than a second.

8.1 Merit

This dissertation research provides a number of merits for the field of software engineering. We developed a new technique for assisting software developers in their attempts to locate faults in programs that is both effective and efficient. Our technique can use commonly available dynamic information that is available from many current testing tools.

We developed a new techniques for locating faults in programs that contain multiple faults. We showed that these techniques are both effective and efficient.

We presented the first work on a new mode of debugging, called parallel debugging, and provided supporting techniques that automate much of this process. We showed that this mode of debugging can provide a significant savings in terms of the time to debug a program.

This research provides a number of directions for future research, and in fact, several researchers have already used our research as a foundation for theirs. Researchers have extended the concepts of our fundamental fault-localization algorithm to new forms of fault localization, re-interpreted our fault-localization metrics in terms of data-mining concepts, re-interpreted and extended our fault-localization metrics in terms of set-similarity concepts, and extended our visualization approaches to three-dimensional displays.

8.2 Future Work

This dissertation research leads to many possible future research direction . The rest of this chapter discusses the future work in three areas.

8.2.1 Exploration of Extensions to Fault Localization Technique

This dissertation has presented a foundational fault-localization technique that can be extended and varied in a number of ways. First, this work can be applied to a number of different coverage entities, such as statements, branches, method calls, and methods. Because one of our goals was to use commonly available testing information, we focused our empirical evaluation on statement-level instrumentation. Future researchers can evaluate whether other forms of instrumentation may prove to be more effective for some types of faults. The type of coverage entity may affect which types of faults are found most effectively. Future researchers can explore the correlation of the type of fault and the method to find them best.

8.2.2 Hierarchical Fault Localization

This dissertation demonstrates that fault localization performed using statement coverage information can be both effective and efficient. However, less efficient techniques may be found that can locate faults more effectively in some cases. For example, the Tarantula technique might be applied to definition-use pairs or sub-paths through the program. Because such instrumentation may be expensive both in terms of the execution overhead time and storage space, these approaches may be less efficient. Techniques could be developed that leverage more efficient approaches to inform and target more expensive approaches. Specifically, a light-weight approach, such as Tarantula using statement coverage, could be used to select regions of the code on which to selectively apply the more expensive approaches. In this way, a hierarchical approach to fault localization could be developed.

8.2.3 Fully Parallelized Debugging

This dissertation presents a foundation for future work on debugging in parallel. This work may motivate a number of future research directions. Here, I identify three areas for potential future research. First, during parallel debugging, one developer could

finish his debugging while another developer is still debugging. In this situation, fixes could be distributed to other developers, or one fault fix may affect the debugging efforts of another developer. Techniques could be developed that automatically provide recommendations in such situations.

Second, organizational and situational constraints will likely dictate the best way to debug in parallel. For example, an imminent release date may require a more aggressive parallelization if redundant developer work can be afforded in an effort to quickly resolve critical bugs. Also, an organization may have a limited number of developers—the parallelization should take this into account. A cost model could be developed that is informed by the program and test suite as well as organizational constraints to customize the technique.

Third, assignment of suspected faults and specialized test suites to the developers that will debug them can be automated. Based on information such as source-code revision history, ownership or familiarity of the suspected faulty code can be mapped to developers. Techniques could be developed to leverage this information to automatically assign developers to fault-localization results and specialized test suites when multiple faults can be debugged simultaneously.

8.2.4 Integrated Fault Localization

This dissertation presents fault-localization techniques that utilize artifacts from the testing process to suggest developer actions with a symbolic debugger. These steps currently involve separate tools and thus require the developer to switch between them. A typical process would require the developer to perform the following three steps: (1) test the program to generate the passed and failed test cases, along with the execution information about each test case; (2) input this testing information to the fault-localization tool which produces information that suggests possible faults in the program; and (3) use a symbolic debugger to place break-points and examine

the state at the suggested faulty sites. The integration of these three tools can further automate much of the developer's work. A tool could run the test suite, automatically find likely faults, rerun test cases, and sample the state at those sites. A debugger could automatically place break-points at likely fault sites. Also, techniques could be developed to integrate source-code management systems that can remember past fault locations and their fixes to potentially inform future debugging efforts. These integrations offer possibilities for both further automation and avenues for easy adoption in practice.

REFERENCES

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

[2] Agrawal, H., *Toward automatic debugging of Computer Programs*. PhD thesis, Purdue University, 1991.

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

[4] Agrawal, R., Imielinski, T., and Swami, A., "Mining association rules between sets of items in large databases," in *Proceedings of ACM SIGMOD International Conference of Data*, pp. 207–216, 1993.

[5] Anvik, J., Hiew, L., and Murphy, G. C., "Who should fix this bug?," in *Proceeding of the International Conference on Software Engineering*, pp. 361–370, May 2006.

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

[7] Balcer, M., Hasling, W., and Ostrand, T., "Automatic generation of test scripts from formal test specifications," in *Proceedings of the Third Symposium on Software, Testing, Analysis, and Verification*, pp. 210–218, December 1989.

[8] Ball, T. and Eick, S. G., "Software visualization in the large," *Computer*, vol. 29, pp. 33–43, Apr. 1996.

[9] Balzer, R. M., "Exdams: Extendible debugging and monitoring system," in *AFIPS Proceedings, Spring Joint Conference*, vol. 34, (Montvale, New Jersey), pp. 567–580, 1969.

[10] Bouthier, C., "Treemap java library," 2002. `http://treemap.sourceforge.net/`.

[11] Bowring, J. F., *Modeling and Predicting Software Behavior*. PhD thesis, Georgia Institute of Technology, 2006.

[12] Bowring, J. F., Rehg, J. M., and Harrold, M. J., "Active learning for automatic classification of software behavior," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 195–205, ACM Press, July 2004.

[13] Briand, L. C., Labiche, Y., and Liu, X., "Using machine learning to support debugging with tarantula," Tech. Rep. SCE-07-04 Version 2, Carleton University, February 2007.

[14] Bruls, M., Huizing, K., and van Wijk, J. J., "Squarified treemaps," in *Proceedings of the Joint Eurographics and IEEE TCVG Symposium on Visualization*, pp. 33–42, 2000.

[15] Bullseye Testing Technology, "BullseyeCoverage," 2007. http://www.bullseye.com/.

[16] Chawla, A. and Orso, A., "A generic instrumentation framework for collecting dynamic information," in *Online Proceedings of the ISSTA Workshop on Empirical Research in Software Testing (WERST 2004)*, (Boston, MA, USA), July 2004.

[17] Cleve, H. and Zeller, A., "Locating causes of program failures," in *Proceedings of the International Conference on Software Engineering*, (St. Louis, Missouri), pp. 342–351, May 2005.

[18] Collofello, J. S. and Cousins, L., "Towards automatic software fault location through decision-to-decision path analysis," in *AFIPS Proceedings of 1987 National Computer Conference*, pp. 539–544, June 1987.

[19] Collofello, J. S. and Woodfield, S. N., "Evaluating the effectiveness of reliability-assurance techniques," *Journal of Systems and Software*, vol. 9, no. 3, pp. 191–195, 1989.

[20] DeMillo, R. A., Pan, H., and Spafford, E. H., "Critical slicing for software fault localization.," in *International symposium on Software testing and analysis*, pp. 121–134, 1996.

[21] Denmat, T., Ducassé, M., and Ridoux, O., "Data mining and cross-checking of execution traces: A re-interpretation of jones, harrold and stasko test information visualization," in *Proceedings of the Conference on Automated Software Engineering*, pp. 396–399, November 2005.

[22] Dickinson, W., Leon, D., and Podgurski, A., "Finding failures by cluster analysis of execution profiles," in *Proceedings of the International Conference on Software Engineering*, pp. 339–348, May 2001.

[23] Doliner, M., "Cobertura," 2006. http://cobertura.sourceforge.net/.

[24] Duda, R. O., Hart, P. E., and Stork, D. G., *Pattern Classification*. John Wiley and Sons, Inc., 2001.

[25] Eclipse Foundation, "Eclipse - an open development platform," 2008. http://www.eclipse.org.

[26] Edison Design Group, "EDG C++ compiler front end," 2007. http://www.edg.com/index.php?location=c_frontend.

[27] Eick, S. G., Steffen, Joseph, L., and Sumner Jr., E. E., "Seesoft—A tool for visualizing line oriented software statistics," *IEEE Transactions on Software Engineering*, vol. 18, pp. 957–968, November 1992.

[28] Elbaum, S., Malishevsky, A., and Rothermel, G., "Prioritizing test cases for regression testing," in *Proceedings of the ACM International Symposium on Softw. Testing and Analysis*, pp. 102–112, Aug. 2000.

[29] Francel, M. A. and Rugaber, S., "The value of slicing while debugging," *Science of Computer Programming*, vol. 40, pp. 151–169, 1996.

[30] Free Software Foundation, "GCC, the GNU Compiler Collection," 2007.

[31] Free Software Foundation, "Gcov—a Test Coverage Program," 2007. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[32] Free Software Foundation, "GDB: The GNU Project Debugger," 2007. http://www.gnu.org/software/gdb/.

[33] Gyimothy, T., Beszedes, A., and Forgacs, I., "An efficient relevant slicing method for debugging," in *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, (London, UK), pp. 303–321, Springer-Verlag, 1999.

[34] H.Agrawal, Demillo, R. A., and Spafford, E. H., "An execution backtracking approach to program debugging," *IEEE Software*, vol. 8, pp. 21–26, May 1991.

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

[36] Harrold, M. J., Gupta, R., and Soffa, M. L., "A methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering and Methodology*, vol. 2, pp. 270–285, July 1993.

[37] Hutchins, M., Foster, H., Goradia, T., and Ostrand, T., "Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proceedings of the International Conference on Software Engineering*, (Sorrento, Italy), pp. 191–200, May 1994.

[38] IBM Rational, "Rational PureCoverage," 2007. http://www-306.ibm.com/software/rational/.

[39] IEEE, "IEEE standard glossary of software engineering terminology," *IEEE Std 610.12-1990*, 1990.

[40] Jcoverage Ltd., "Jcoverage," 2008. http://www.jcoverage.com/.

[41] Jones, J. and Harrold, M. J., "Test-suite reduction and prioritization for modified condition/decision coverage," in *Proceedings of the International Conference on Software Maintenance*, pp. 92–101, November 2001.

[42] Jones, J., Harrold, M. J., and Stasko, J., "Visualization of test information to assist fault localization," in *Proceedings of the International Conference on Software Engineering*, (Orlando, Florida), pp. 467–477, May 2002.

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

[44] Jones, J. A. and Harrold, M. J., "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, (Long Beach, California, USA), pp. 273–282, November 2005.

[45] Korel, B. and Laski, J., "Dynamic program slicing," *Information Processing Letters*, vol. 29, pp. 155–63, Oct. 1988.

[46] Lamping, J., Rao, R., and Pirolli, P., "A focus+context technique based on hyperbolic geometry for visualizing large hierarchies," in *Proceedings of the Conference on Human Factors in Computing Systems*, pp. 401–408, 1995.

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

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

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

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

[51] Lukey, F. J., "Understanding and debugging programs," *International Journal of Man-Machine Studies*, vol. 12, pp. 189–202, February 1980.

[52] Microsoft Corporation, "Microsoft Visual Studio," 2008. http://msdn2.microsoft.com/en-us/vstudio/default.aspx.

[53] National Institute of Standards and Technlogy (NIST), Department of Commerce, "Software errors cost U.S. economy $59.5 billion annually," *NIST News Release 2002-10*, 2002. http://www.nist.gov/public_affairs/releases/n02-10.htm.

[54] Ostrand, T. and Balcer, M., "The category-partition method for specifying and generating functional tests," *Communications of the ACM*, vol. 31, June 1988.

[55] Pan, H. and Spafford, E., "Heuristics for automatic localization of software faults," Tech. Rep. SERC-TR-116-P, Purdue University, 1992.

[56] Pan, H., DeMillo, R. A., and Spafford, E. H., "Failure and fault analysis for software debugging," in *Proceedings of COMPSAC 97*, (Washington, D.C.), pp. 515–521, August 1997.

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

[58] Postel, J. B., "RFC821: Simple Mail Transfer Protocol," 1982. `http://www.ietf.org/rfc/rfc0821.txt`.

[59] Ren, X., Ryder, B. G., Stoerzer, M., and Tip, F., "Chianti: a change impact analysis tool for java programs," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pp. 664–665, 2005.

[60] Renieris, E., *A Research Framework for Software-Fault Localization Tools*. PhD thesis, Brown University, 2005.

[61] Renieris, E. and Reiss, S., "Fault localization with nearest neighbor queries," in *Proceedings of the International Conference on Automated Software Engineering*, (Montreal, Quebec), pp. 30–39, October 2003.

[62] Renner, S., "Location of logical errors on pascal programs with an appendix on implementation problems in waterloo prolog/c," Tech. Rep. UIUCDCS-F-82-896, University of Illinois at Urbana-Champaign, Urbana, Illinois, April 1982.

[63] Rothermel, G. and Harrold, M. J., "Empirical studies of a safe regression test selection technique," *IEEE Transactions on Software Engineering*, vol. 24, pp. 401–419, 1998.

[64] Rothermel, G., Harrold, M. J., Ostrin, J., and Hong, C., "An empirical study of the effects of minimization on the fault detection capabilities of test suites," in *Proceedings of the International Conference on Software Maintenance*, Nov. 1998.

[65] Rothermel, G., Untch, R., Chu, C., and Harrold, M. J., "Test case prioritization: An empirical study," in *Proc. of the Int'l Conf. on Softw. Maint.*, pp. 179–188, September 1999.

[66] Rothermel, G., Untch, R., Chu, C., and Harrold, M. J., "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, pp. 929–948, October 2001.

[67] Sedlmeyer, R. L., Thompson, W. B., and Johnson, P. E., "Knowledge-based fault localization in debugging," in *SIGSOFT '83: Proceedings of the symposium on High-level debugging*, (New York, NY, USA), pp. 25–31, ACM Press, March 1983.

[68] Shapiro, E. Y., *Algorithmic Program Debugging*. PhD thesis, Yale University, Cambridge, Massachusetts, 1983.

[69] Shneiderman, B., "Tree visualization with tree-maps: A 2-d space-filling approach," *ACM Transactions on Computer Graphics*, vol. 11, pp. 92–99, Jan. 1992.

[70] Stasko, J. and Zhang, E., "Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations," in *Proceedings of the IEEE Symposium on Information Visualization*, pp. 57–65, 2000.

[71] Sun Microsystems, "Debugging a Program With dbx," 2007. http://docs.sun.com/app/docs/doc/819-5257.

[72] University of Washington, "The IMAP Connection," 2002. `http://www.imap.org/`.

[73] Vessey, I., "Expertise in debugging computer programs," *International Journal of Man-Machine Studies: A process analysis*, vol. 23, no. 5, pp. 459–494, 1985.

[74] Vokolos, F. and Frankl, P., "Empirical evaluation of the textual differencing regression testing tec hniques," in *International Conference on Software Maintenance*, November 1998.

[75] Weiser, M., "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, pp. 446–452, July 1982.

[76] Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, pp. 352–357, July 1984.

[77] Zeller, A., "DDD: Data Display Debugger," 2006. http://www.gnu.org/software/ddd/.

[78] Zhang, X., Gupta, N., and Gupta, R., "Locating faults through automated predicate switching," in *ACM/IEEE International Conference on Software Engineering*, pp. 272–281, May 2006.

[79] Zheng, A., Jordan, M. I., Liblit, B., Naik, M., and Aiken, A., "Statistical debugging: Simultaneous identification of multiple bugs," in *Proceedings of the International Conference on Machine Learning*, pp. 1105–1112, June 2006.

VITA

James was born in 1972 and was raised in Avon Lake, Ohio, USA. He attended Ohio State University and in 1996 received the B.S. in Computer Science (Summa Cum Laude). After graduation, he worked at Ohio State as a Systems Engineer/Developer developing software-analysis tools to support research in the area. In 2000, he took an assignment at Georgia Institute of Technology as a Research Scientist, and began his Ph.D. studies that Fall under the advisement of Mary Jean Harrold. He will receive the Ph.D. in Computer Science in April of 2008.