

Fault Localization for Exploratory Simulations

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Ross Gore

August 2012

Abstract

Predicate-level statistical debuggers are applied to software to identify program statements which cause software failures. These debuggers use a set of conditional propositions, or predicates, which are inserted into a program and tested at particular points. The debugger estimates the likelihood that each predicate reflects the fault via suspiciousness. Then, developers examine the predicates in decreasing order of suspiciousness estimate until the location of the fault is discovered.

While predicate-level statistical debuggers are often effective, they are not well suited to an important class of software, including exploratory simulations, which employ floating-point computations and continuous stochastic distributions to represent, or support the evaluation of, an underlying model. This dissertation addresses this deficiency with three contributions to improve the effectiveness of predicate-level statistical debuggers for software employing floating point computations and continuous stochastic distributions.

First, the concept of elastic predicates is introduced and their viability and effectiveness are established. Elastic predicates are formed from summary statistics of variable values profiled at instrumented program points. The result is predicates that more closely match where a fault is expressed. Experimental results for established benchmarks and widely used simulations show improved effectiveness and reasonable efficiency with elastic predicates.

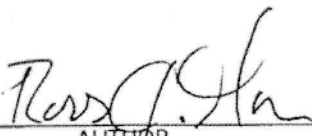
Next, confounding bias within the measures used to estimate predicate suspiciousness is addressed. Confounding bias is reduced via a model that controls for two different previously unaccounted for variables that influence the suspiciousness of a predicate. Statistical matching is employed to further reduce confounding effects. Empirical results show that matching, in conjunction with the new model, significantly improves the effectiveness of the suspiciousness estimate in predicate-level statistical debuggers, including those employing elastic predicates.

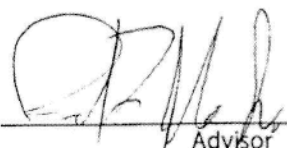
Finally, the first analytical result concerning many-valued, as opposed to binary, passing extents for statistical debuggers is presented. These continuous or fuzzy, passing extents enable debugging to be applied to a large class of software employing stochastic distributions, which previously had been inapplicable for statistical debugging.

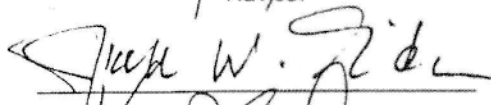
Combined, elastic predicates, reduced bias suspiciousness estimates and fuzzy passing extents significantly improve the state of the art for localizing sources of failures in all software, but particularly software employing floating point computations and continuous stochastic distributions.

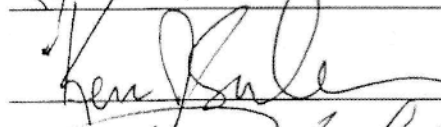
APPROVAL SHEET

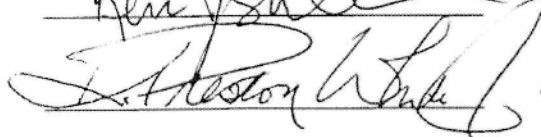
is submitted in partial fulfillment of the requirements
for the degree of

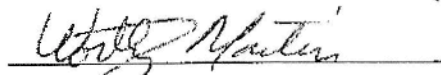

AUTHOR

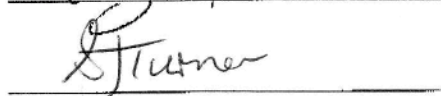

Advisor



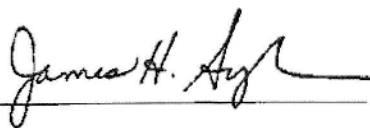








Accepted for the School of Engineering and Applied Science:



Dean, School of Engineering and Applied Science

To everyone who has helped me succeed.

Acknowledgements

I am grateful to my parents for instilling in me the value of education. This dissertation would not have been possible without their endless encouragement and sacrifices. Their confidence and pride in my achievements have helped carry me through numerous difficult times. I thank them for providing an excellent family environment and for supporting me in various life decisions.

To Paul, my advisor and friend, I express my sincere appreciation for his invaluable encouragement throughout my years at UVa. His support, creativity and ability to work on multiple problems at once have been inspirational. I am grateful to him for educating me on the various aspects of being a good researcher. I thank him for the countless hours spent discussing non-academic matters of life. I treasure this opportunity I have had to work with and look forward to many more.

Words cannot express how much the love and unfailing confidence of my fiancée, Sarah, have meant to me. I thank her for always being there for me and for being the stabilizing force in my life. I am grateful to my brother, Gavin, for his continued support. I am incredibly fortunate to have a brother who is also one of my closest friends.

The many meetings with the Modeling and Simulation Technology and Research Initiative group have always been productive for me. I thank Michael Spiegel and David Kamensky for numerous exciting discussions - it was a pleasure working with both of them.

I thank my committee members Professors Worthy Martin, Kevin Sullivan, Jack Davidson, Preston White and Stephen Turner for their insights and suggestions about my research. I also thank the faculty of the Computer Science Department for their support.

I am especially thankful to Scott Ruffner and Rick Stillings for helping me out with the computer systems many times over the years. Also, I am grateful to Brenda Perkins, Wendy Morris and Kim Gregg for their invaluable assistance with many bureaucratic aspects of graduate life.

I thank my roommates Kyle Haynes and Dan Williams for keeping the hours at home lively. Their friendship and ability to empathize with me has helped keep me sane during difficult times in my graduate career.

Contents

Contents	vi
List of Tables	viii
List of Figures	x
1 Introduction	1
1.1 Elastic Predicates	2
1.2 Reducing Confounding Bias in Predicate-Level Debuggers	4
1.3 Fuzzy Passing Extents	5
1.4 Measuring Success	6
1.5 Summary	6
2 Related Work	8
2.1 Statistical Approaches	8
2.1.1 Statement-level Statistical Approaches	9
2.1.2 Predicate-level Statistical Approaches	10
2.2 State-altering Approaches	11
2.3 Program Slicing Approaches	12
2.3.1 Static Program Slicing	12
2.3.2 Dynamic Slicing	13
2.4 Other Approaches to Fault Localization	14
2.5 Failure Specific Approaches	15
2.6 Static Approaches	16
2.7 Casual Inference	16
2.7.1 Causal Graphs and Models	17
2.7.2 Causal Estimation	17
2.8 Summary	20
3 Elastic Predicates	21
3.1 Motivating Example	22
3.2 Suspiciousness Estimates	24
3.2.1 Specificity	24
3.2.2 Tarantula	24
3.2.3 F_1 Measure	25
3.2.4 Importance	25
3.2.5 Ochiai	26
3.3 Methodology	26
3.3.1 Single Variable	27
3.3.2 Scalar Pairs	28
3.3.3 Instrumentation	29
3.3.4 Inferred Predicates	30
3.4 Evaluation	30
3.4.1 Subject Programs	31
3.4.2 Competing Fault Localization Approaches	31

3.4.3	Effectiveness Studies	34
3.4.4	Widely Used Simulations	36
3.4.5	Sparse Sampling	38
3.4.6	Incomplete Test Suites	39
3.4.7	Suspiciousness Maximized Elastic Predicates	41
3.4.8	Efficiency	43
3.4.9	Reduced Elastic Predicates	45
3.4.10	Validity	50
3.4.11	Multiple Faults	50
3.5	Chapter Summary	51
4	Reducing Confounding Bias	52
4.1	Motivating Example	52
4.2	Controlling for Confounding Bias	55
4.2.1	Observational Studies, Causal Models and Confounding Bias	56
4.2.2	Control Flow Dependency Confounding Bias	58
4.2.3	Failure Flow Confounding Bias	60
4.2.4	Improving Existing Suspiciousness Estimates	61
4.2.5	Evaluation	62
4.3	Matching	73
4.3.1	Motivating Example	73
4.3.2	Exact Matching	75
4.3.3	Mahalanobis Distance Matching	76
4.4	Evaluation	78
4.5	Chapter Summary	85
5	Fuzzy Passing Extents	87
5.1	Fuzzy Logic Background	88
5.2	Fuzzy Passing Extents	89
5.3	Fuzzy Passing Extent Example	92
5.4	Evaluation	94
5.4.1	Subject Programs	94
5.4.2	Competing Statistical Debuggers	97
5.4.3	Effectiveness	98
5.4.4	Efficiency	101
5.4.5	Validity	103
5.5	Chapter Summary	103
6	Discussion	105
6.1	Effectiveness	105
6.2	Overhead	110
6.3	Time Incurred	112
6.4	Summary	117
7	Conclusion	118
7.1	Summary of Contributions	118
7.2	New Research Areas	120
7.3	Availability of ESP	121
7.4	Future Work	121
7.5	Final Remarks	122
	Bibliography	124

List of Tables

3.1	Fundamental single variable elastic predicates.	27
3.2	The top ranked predicates for Figure 3.1.	28
3.3	Fundamental scalar pairs elastic predicates.	29
3.4	Subject programs used in the evaluation of elastic predicates	32
3.5	Number (percentage) of faulty version ranked statement lists in each score range for all approaches for all subject programs.	34
3.6	Rank of faulty statement for all approaches for the widely used simulations.	37
3.7	Number (percentage) of faulty version ranked statement lists in each score range for CBI, ESP and MAX.	43
3.8	Wallclock time (in seconds) required by each approach to execute all of the faulty versions of the specified subject program.	45
3.9	Groups of elastic predicates in ESP with similar suspiciousness estimates.	46
3.10	FAST ESP elastic predicates.	46
3.11	Number (percentage) of faulty version ranked statement lists in each score range for FAST ESP and ESP.	47
3.12	Wallclock time (in seconds) required by ESP and FAST ESP to execute all of the faulty versions of the specified subject program.	48
4.1	Test cases and predicate data for procedure <code>distance()</code> in Figure 4.1.	54
4.2	Most suspicious predicates by <i>specificity</i> estimate for the procedure <code>distance()</code> in Figure 4.1.	54
4.3	Execution data for predicate <code>(need_upward_RA > 0)</code> ₁₂₆ in Figure 4.9.	74
4.4	Comparison of matching techniques within CBI.	80
4.5	Comparison of matching techniques within ESP.	80
4.6	Wallclock time (in seconds) required by each approach to execute all the faulty versions of the specified subject program.	83
5.1	The passing function used for <code>tcas</code> test cases.	95
5.2	Subject programs used in the evaluation of fuzzy passing extents	95
5.3	Measures used by the traditional and fuzzy versions of Tarantula	98
5.4	Number (percentage) of faulty version ranked statement lists in each score range for all approaches.	99
5.5	Average wallclock time (in seconds) required by each approach to execute all the faulty versions of the specified subject program over 100 trials.	101
6.1	Number (percentage) of faulty version ranked statement lists in each score range for all approaches.	106
6.2	Number (percentage) of faulty version ranked statement lists in each score range for all approaches for the programs which employ floating-point computations and continuous stochastic distributions.	108
6.3	Mean and median <i>Cost</i> for different subject program sizes.	108
6.4	Number (percentage) of faulty version ranked statement lists in each <i>Time Incurred</i> range for all approaches for all programs included in the evaluation.	113

6.5	Number (percentage) of faulty ranked statement lists in each <i>Time Incurred</i> range for all approaches for the programs which employ floating-point computations and continuous stochastic distributions.	113
7.1	Publications featuring the contributions of this dissertation.	123

List of Figures

2.1	Causal graph illustrating <i>Back-Door Criterion</i>	19
3.1	The source code for the <code>more_arrays()</code> function in BC.	23
3.2	Comparison of the effectiveness of all the approaches. Higher and further left is better.	35
3.3	CBI (non-striped) and ESP (striped) under sparse sampling rates for the subject programs.	39
3.4	CBI (non-striped) and ESP (striped) with incomplete test suites for the subject programs.	40
3.5	Comparison of the effectiveness of CBI, ESP and MAX. Higher and further left is better.	42
3.6	<i>Slowdown</i> relative to Tarantula for each competing approach. Note the log scale of the y-axis.	44
3.7	Comparison of the effectiveness of ESP and FAST ESP. Higher and further left is better.	47
3.8	Comparison of the efficiency of ESP and FAST ESP. Note the log scale of the y-axis.	49
4.1	Faulty function that calculates euclidean distance between two one-dimensional points	53
4.2	Improvements in effectiveness achieved in ESP (red) and CBI (white) by using $\tau_{ls,p}^c$ as the suspiciousness estimate instead of the standard <i>Tarantula</i> estimate.	64
4.3	Improvements in effectiveness achieved in ESP (red) and CBI (white) by using $\tau_{ls,p}^{c,f}$ as the suspiciousness estimate instead of $\tau_{ls,p}^c$	65
4.4	Improvements in effectiveness achieved in ESP (red) and CBI (white) by integrating $\tau_{ls,p}^c$ into the F_1 suspiciousness estimate.	67
4.5	Improvements in effectiveness achieved in ESP (red) and CBI (white) by integrating $\tau_{ls,p}^{c,f}$ into the F_1 suspiciousness estimate instead of $\tau_{ls,p}^c$	68
4.6	Improvements in effectiveness achieved in ESP (red) and CBI (white) by integrating $\tau_{ls,p}^c$ into the <i>Ochiai</i> suspiciousness estimate.	69
4.7	Improvements in effectiveness achieved in ESP (red) and CBI (white) by integrating $\tau_{ls,p}^{c,f}$ into the <i>Ochiai</i> suspiciousness estimate instead of $\tau_{ls,p}^c$	70
4.8	Improvements in effectiveness achieved in ESP (red) and CBI (white) by integrating $\tau_{ls,p}^{c,f}$ into the Importance suspiciousness estimate instead of the difference heuristic: $f_p/(f_p + s_p) - f_{p\text{ obs}}/(f_{p\text{ obs}} + s_{p\text{ obs}})$	71
4.9	An excerpt from a faulty implementation of <code>tcas</code>	74
4.10	The pairing of test cases in <i>MD matching</i> between control and treatment groups which reduces <i>complete lack of overlap</i>	77
4.11	Effectiveness of MDM vs NM in ESP (red) and CBI (white).	81
4.12	<i>Slowdown</i> of MDM versions of ESP and CBI compared to NM versions. Note the log scale of the y-axis.	84
5.1	Graph of the age scale for the descriptions <i>young</i> , <i>middle-aged</i> , <i>old</i>	89
5.2	Evaluation of the effectiveness of <i>fuzzy passing extents</i> in Tarantula, CBI and ESP.	99
5.3	Evaluation of the efficiency of <i>fuzzy passing extents</i> in Tarantula, CBI and ESP. Note the log scale of the y-axis.	102
6.1	Comparison of the effectiveness of Tarantula, CBI and ESP using the <i>Cost</i> metric. Higher and further left is better.	107

6.2	Comparison of the effectiveness of Tarantula, CBI and ESP using the <i>Cost</i> metric for the programs included in the evaluation which employ floating-point computations and continuous stochastic distributions. Higher and further left is better.	109
6.3	Graph of wallclock time overhead (in %) incurred by CBI when employing each contribution in this dissertation.	110
6.4	Comparison of the effectiveness of Tarantula, CBI and ESP using the <i>Time Incurred</i> metric. Higher and further left is better.	115
6.5	Comparison of the effectiveness of Tarantula, CBI and ESP using the <i>Time Incurred</i> metric for the programs included in the evaluation which employ floating-point computations and continuous stochastic distributions. Higher and further left is better.	116

Chapter 1

Introduction

When a failure is first observed in software, the prospect of localizing the fault causing it can be daunting. A software *fault* is an invalid token or bag of tokens in a program statement that will cause a failure when the compiled code for the program which implements the source code is executed [1]. A software *failure* occurs when the output of an executed test case, supplied for the program, does not produce the output specified by the requirements [2].

Common practice is to apply classic debugging techniques to identify the program statements and interactions that lead to the fault. This practice is largely manual and it can consume weeks, months and even years of effort. Resources in terms of money, people and time should be minimized in the process of fault localization [3, 4, 5, 6, 7, 8]. Predicate-level statistical debuggers have been employed to meet this goal [9, 10, 11, 12, 13, 14, 15, 16, 17].

Predicate-level statistical debuggers use a set of conditional propositions, or predicates, which are inserted into a program and tested at particular points. Each debugger estimates the likelihood that the inserted predicates reflect the fault via suspiciousness. The suspiciousness estimate is determined, in part, by the ratio of: (a) test cases that include the predicate and produce the specified output (or pass) to (b) test cases that include the predicate and do not produce the specified output (or fail). Developers examine the predicates in decreasing order of suspiciousness estimate until the location of the fault is discovered [9, 10, 11].

While predicate-level statistical debuggers and other automated fault localization approaches are effective in general, they are not tailored to an important class of software, including simulations and computational models, which employ floating-point computations and continuous stochastic distributions. Existing predicate-level statistical debugging approaches do not employ predicates for floating-point computations because of the additional amount of space and time floating-point variables require compared to character, Boolean,

integer and pointer types [10]. Furthermore, the use of stochastic distributions defies the assumptions made by other fault localization approaches which support floating-point computations. These approaches assume subject program execution traces and repeatable outcomes, which is not true for subject programs employing stochastics [18, 19, 20, 21, 22, 23, 24].

In this dissertation these fault localization deficiencies are addressed. The thesis framing this pursuit is:

Thesis: *Practically significant* improvements in terms of effectiveness and applicability can be made to fault localization through the introduction of novel program-specific predicates, improved predicate suspiciousness estimation and many-valued, as opposed to binary, passing extents.

Two key insights underlie this hypothesis. The first is that additional data can be profiled from subject program executions to enable more effective fault localization analysis than can be done with existing approaches. This additional data can: (1) create novel program specific predicates which more accurately capture where a fault is and is not expressed and (2) account for factors which create confounding bias in the measures used to estimate the suspiciousness of instrumented predicates. The second insight is that user knowledge of the subject program specification can create continuous functions which specify the extent to which a subject program execution passes (or fails) a test case. These functions enable the program specific predicates and reduced bias suspiciousness estimates, along with previously published statistical debugging contributions, to be applied to programs employing floating-point computations and continuous stochastic distributions, which had been previously outside the scope of fault localization. The term *practical significance* is used in the thesis to indicate that the improvements made to fault localization will be significant in magnitude and have an observable effect for end users [25].

We begin by introducing the three research contributions that result from these insights. Each contribution reflects a chapter in the dissertation. An overview of our approach to evaluating if these contributions are sufficiently efficient and effective is presented in Section 1.4. Measures of *efficiency* quantify the wallclock time required by a debugger to produce a ranked list of predicates for a faulty subject program. Measures of *effectiveness* quantify how well the ranked predicates localize the fault in the subject program.

1.1 Elastic Predicates

Predicates in existing statistical debuggers are *static* in the sense that the conditional propositions tested in the debugger are specified before the subject program is executed. For example, in the canonical predicate-level statistical debugger, Cooperative Bug Isolation (CBI), three static predicates are tested for each assignment to a variable x in a statement y . These predicates are: $(x_y > 0)$, $(x_y = 0)$ and $(x_y < 0)$ [9, 10, 11]. The

decision to compare the value of x_y to 0 in each of these predicates is made before the subject program is executed, making these predicates *static*.

While static predicates have been shown to be effective for a variety of general-purpose software applications, they can be improved for applications employing floating-point computations and continuous stochastic distributions [26]. Frequently in these programs all the values assigned to a given variable satisfy the same static predicate. The result is poor fault localization analysis. This issue can be addressed by profiling the values assigned to variables during subject program executions.

Ideally, the value assigned to each variable in a subject program execution would be stored to create a program specific, or *elastic predicate*, that better localizes faults. Given a measure to estimate predicate suspiciousness, a *maximized elastic predicate* would identify the values assigned to a variable x_y that maximize the suspiciousness of the predicate. A *maximized elastic predicate* would be the most effective predicate constructed of continuous values for fault localization because it identifies the most suspicious range of values assigned to each x_y .

Unfortunately, generating *maximized elastic predicates* is infeasible in practice. *Optimal elastic predicates* require all of the values assigned to each x_y to be stored and sorted. For many subject programs, the space required to store all of the values assigned to each x_y will exceed the space available on a modern workstation. Furthermore, sorting the assigned values to each x_y cannot be performed quickly enough to make the resulting predicate useful.

In Chapter 3 we investigate if the benefits of *elastic predicates* are realizable in practice. First, a set of *elastic predicates* that can be computed efficiently in terms of time and space is presented. These *elastic predicates* use summary statistics such as the mean, μ_{x_y} , and standard deviation, σ_{x_y} , of the values assigned to a variable x in a statement y to cluster together similar values. The methodology used to instrument faulty subject programs with our elastic predicates in faulty subject programs is also described.

Next, the efficiency and effectiveness of our *elastic predicates* is thoroughly evaluated. First, the evaluation tests how much more effective a predicate-level statistical debugger employing both static and *elastic predicates* is compared to a predicate-level statistical debugger that only employs static predicates. One portion of this evaluation focuses on only localizing faults in software employing floating-point computations and continuous stochastic distributions. Second, the evaluation tests how much of the effectiveness provided by *maximized elastic predicates* our *elastic predicates* realize. This evaluation is performed for a set of fault localization benchmarks where *maximized elastic predicates* can be computed. Finally, the evaluation tests how the uncertainty introduced through sparse random sampling of predicates and incomplete test suites reduces the effectiveness of elastic predicates.

Evidence gathered from the evaluation demonstrates that our *elastic predicates* significantly improve the effectiveness of statistical debuggers for a variety of existing fault localization benchmarks and a set of programs employing floating-point computations and continuous stochastic distributions. While these predicates do not fully realize all the benefits of *maximized elastic predicates* they account for a substantial amount of the improvements in effectiveness that are possible. Furthermore, evidence from the evaluation shows that these improvements persist in the face of sparse sampling rates and incomplete test suites.

1.2 Reducing Confounding Bias in Predicate-Level Debuggers

The effectiveness of predicate-level statistical debuggers can be improved through multiple avenues. The *elastic predicates* presented in Chapter 3 improve statistical debuggers by creating program specific predicates that more effectively localize causes of failure in these programs. However, statistical debuggers can also be made more effective by developing a methodology to estimate the suspiciousness of a predicate more accurately by profiling additional data from subject program executions.

In Chapter 4 we investigate what data needs to be profiled to achieve more accurate suspiciousness estimation of predicates in all programs, including programs employing floating point computations and continuous stochastic distributions. First, the two factors within faulty subject programs which create bias in existing predicate-level suspiciousness estimates are identified. These factors are: (1) control flow dependencies between predicates and (2) statement coverage. Next, a description of the methodology we use to profile subject programs to collect data related to these factors is presented.

Using this profiled data, a model based in the field of observational studies and causal inference which accounts for the effect of control flow dependencies between predicates and statement coverage on predicate suspiciousness estimation is proposed. By accounting for these factors the model is capable of producing a suspiciousness estimate where bias is reduced and/or eliminated.

In its first incarnation, our predicate-level statistical debugger assumed that the patterns of control flow dependencies between predicates and statement coverage were the same in all test cases. We demonstrate that this assumption is not true. This deficiency is addressed by employing statistical matching to identify and pair together those test cases within a test suite that contain similar patterns of control flow dependencies between predicates and statement coverage. These *matched* test cases are used to estimate the suspiciousness of the instrumented predicates with reduced bias. We demonstrate that our reduced bias suspiciousness estimate can be integrated into dominant, existing suspiciousness measures.

Next, the accuracy of the suspiciousness estimate resulting from our model is thoroughly evaluated. First, the evaluation tests the ability of our reduced bias estimate and existing suspiciousness estimates to localize

faults in a set of established benchmarks. Finally, the evaluation tests how the uncertainty introduced through sparse random sampling of predicates affects the accuracy of our reduced bias suspiciousness estimate.

Evidence gathered from the evaluation demonstrates that control flow dependencies between predicates and statement coverage create significant confounding bias in the existing measures used to estimate the suspiciousness of predicates. The bias is manifested because existing measures do not account for these factors. A model based in observational studies and causal inference enables these factors to be accounted for, and their effects on the estimate of a predicate’s suspiciousness can be reduced and/or eliminated. The evaluation shows that this methodology produces more accurate suspiciousness estimates and more effective fault localization. This improved effectiveness is applicable to all software, including programs which employ floating-point computations and continuous stochastic distributions. Furthermore, these improvements persist in the face of sparse predicate sampling rates.

1.3 Fuzzy Passing Extents

Existing approaches to fault localization require a Boolean function to label the output of a program as passing or failing. However, the output of software that employs stochastics can include random variance. Frequently these pieces of software are exploratory simulations, where the precise output for a given test case is not exactly known. In these cases, it is difficult to determine if the simulation passes or fails a given test case. As a result the improvements offered by the *elastic predicates*, the reduced bias suspiciousness estimation and a bevy of other fault localization research contributions are inapplicable to this software.

In Chapter 5 we address this issue with *fuzzy passing extents*, which are inspired by fuzzy logic. A description of how *fuzzy passing extents* enable users of software with variance in the output to define a passing function which specifies the degree $[0,1]$ to which a subject program passes a given test case is provided. The result is that *fuzzy passing extents* enable a subject program execution to both pass and fail a given test case.

An example that demonstrates how a user could define a passing function and employ *fuzzy passing extents* to localize a fault in a canonical Monte Carlo simulation with variance in the output is presented. Furthermore, we demonstrate that it is trivial to define a passing function which results in the same fault localization capabilities as existing debuggers which require a Boolean function to label the output of a program as passing or failing. In this sense, *fuzzy passing extents* are backwards compatible with dominant, existing fault localization tools and suspiciousness estimates.

Next an evaluation of *fuzzy passing extents* is presented. The evaluation employs several existing statistical debuggers that we augmented with *fuzzy passing extents* and compares how well they localize faults in a set

of adapted benchmarks. The adapted benchmarks reflect several established fault localization benchmarks that are modified in a straightforward manner to include variance in their output. Similarly straightforward passing functions are defined for each of the included benchmarks.

Evidence gathered from the evaluation shows that debuggers employing *fuzzy passing extents* are significantly more effective at localizing faults in programs with variance in the output than their traditional counterparts. Furthermore, while the application of *fuzzy passing extents* does not guarantee improved effectiveness for a statistical debugger we demonstrate that they can be made to reproduce the results of the existing fault localization tools and suspiciousness estimates if needed.

1.4 Measuring Success

The long-term goal of our work is to produce automated tools to localize the sources of failures in software employing floating-point computations and continuous stochastic distributions. The contributions which achieve our goal and support the thesis of this dissertation stem from novel fault analysis which incurs additional overhead and reduces efficiency compared to existing approaches. As a result, the evaluation of our contributions takes two forms.

The first form of evaluation ignores efficiency and directly compares the effectiveness of debuggers employing our contributions to existing approaches. These evaluations can be found throughout this dissertation. The second form of evaluation takes into account: (1) the time required to generate and use our contributions and (2) the effectiveness of the fault localization analysis resulting from our contributions. This evaluation tests that the total time required by users to localize faults in subject programs is reduced when using our analysis versus existing tools. This evaluation serves as the overall measure of success for our work and is presented in Chapter 6.

We claim that predicate-level statistical debuggers are deficient for localizing failures in programs employing floating-point computations and continuous stochastic distributions. This dissertation proposes contributions to provide developers with more effective automated debuggers for localizing faults in these programs. Ultimately, the evaluation strategy summarized here and employed throughout this dissertation measures if the proposed contributions are successful in addressing this deficiency and achieving our goal.

1.5 Summary

By all evaluation measures commonly employed in the fault localization community, our work improves fault localization effectiveness not only for subject programs employing continuous stochastic distributions and

floating-point computations, but for a broad range of subject programs that appear in our test suite of over 300 faulty subject program versions.

To achieve these results we incur an approximately 5-fold increase in computational analysis time over the best competing algorithms. In cases where minimization of human analysis time is favored over minimization of computational analysis time, ours is a most effective method. When computational analysis time matters, we have presented ideas for improving the analysis time of our approach and consider further pursuit of the question to be a next logical step beyond the work presented here.

Chapter 2

Related Work

The problem of identifying faults remains a challenge in software engineering. In this section the advantages and drawbacks of existing fault localization approaches are reviewed. First, statistical approaches to fault localization are discussed, followed by state-altering approaches, slicing-based approaches and other dynamic approaches. Then, fault specific approaches and static approaches to fault localization are reviewed. Finally, background material on causal inference needed to understand the approach to fault localization taken in this dissertation is presented.

2.1 Statistical Approaches

Recently, there has been considerable research on using statistical approaches for fault localization. These approaches, referred to as *statistical debuggers*, require test inputs, corresponding execution profiles and a labeling of the test executions as either succeeding or failing. The execution profiles reflect the coverage of program elements. The approaches employ an estimate of suspiciousness to rank the program elements. Developers examine program elements in decreasing order of the *suspiciousness* estimate until the fault is discovered.

Program elements refer to individual program statements or the truth-values of conditional propositions represented by branches or inserted predicates. Statistical approaches at the statement-level are more efficient but less effective than approaches at the predicate-level because they only consider statement coverage and *not* the values of variables within statements. Statement-level statistical approaches are reviewed in 2.1.1, then predicate-level statistical approaches are summarized in 2.1.2.

2.1.1 Statement-level Statistical Approaches

Statement-level statistical fault localization research focuses on the discovery that failing program executions are likely to have different statement coverage patterns compared to passing program executions [3, 4]. Initial research in this area centered around identifying statistics of statement coverage to effectively estimate the suspiciousness of statements. The goal was to create a statistic that maximized the suspiciousness estimate for the program statement containing the fault.

In the *Nearest Neighbor* approach a failing execution is paired with a larger number of passing executions. The passing execution that is most similar to the failing execution is identified and differences between the two are labeled as suspicious [5]. *Set-union* and *Set-intersection* approaches have been explored as well. The *Set-intersection* approach identifies suspicious statements by computing the set difference between the statements that are present in every passed execution and the statements in a single failing execution [6]. The *Set-union* approach identifies suspicious statements by removing the union of all statements in passing executions from the statements within a failing execution [6].

Tarantula, developed by Jones et. al., is the first statistical approach to recognize that statements exercised more often by failing runs than by passing runs, are likely to be faulty [7, 8]. *Tarantula* calculates the fraction of failed executions that execute a statement, over all passed executions. It uses the former fraction over the sum of the two fractions as the statement's suspiciousness estimate.

The *Tarantula* suspiciousness estimate measures *specificity* (or *precision*). Other statement suspiciousness estimates balance *Tarantula*'s specificity measure with a *sensitivity* (or *recall*) measure. For example in *Statement-level Bug Isolation (SBI)*, Yu et al. combine the proportion of executions including the statement that are faulty and the proportion of faulty executions in which the statement appears together via their harmonic mean to yield a suspiciousness estimate balancing both *sensitivity* and *specificity* [27]. Similarly the *Ochiai* measure balances *sensitivity* and *specificity* via their geometric mean [12].

Over time *Tarantula*, *SBI* and other similar approaches were shown to be more effective than *Set-Union*, *Set-Intersection* and *Nearest Neighbor* [7, 27, 12]. As a result, *Tarantula*, *SBI* and other similar approaches have been enhanced. Jones et al. [28] have explored how to cluster test cases in *Tarantula* to facilitate multiple developers to debug a faulty program in parallel. Baudry et al. [29] observed that some groups of statements, referred to as *dynamic basic blocks*, are always executed by the same set of test cases. To optimize *Tarantula*, they find a subset of the original test suite that aims to maximize the number of *dynamic basic blocks* executed.

More recent research has focused on identifying the best measure to estimate statement suspiciousness. The explored measures include F_1 measure (harmonic mean of *specificity* and *sensitivity*) [27], *Tarantula*

suspiciousness (*specificity*) [7, 8, 28], capture propagation (*specificity* of basic blocks) [30] and the *Ochiai* measure (geometric mean of *sensitivity* and *specificity*) [12]. Research has shown that in practice the most effective estimate of statement suspiciousness is the *Ochiai* measure [12].

However, even the *Ochiai* measure can be further improved. Baah et al. showed that all statement-level suspiciousness estimates contain confounding bias due to subject program control and data flow dependencies. Baah et al. propose a causal model at the statement-level that reduces control and data flow dependency confounding bias, improving the effectiveness of the resulting suspiciousness estimates and the fault localization tools that employ them [31, 32].

2.1.2 Predicate-level Statistical Approaches

Predicate-level statistical debugging approaches represent a class of fault localization techniques that share a common structure. Each approach consists of a set of conditional propositions, or predicates, which are inserted into a program and tested at particular points. A single predicate can be thought of as partitioning the space of all test cases into two subspaces: those satisfying the predicate and those not. Better predicates create partitions that more closely match the space where the fault is expressed. Similar to statements, the predicates are ranked, based on their estimated suspiciousness and guide developers in finding and fixing faults.

The canonical predicate-level statistical debugger *Cooperative Bug Isolation (CBI)* [9, 10, 11] computes the proportion of failing executions where a predicate is evaluated (true or false) and the proportion of failing executions where a predicate is evaluated and true. It then calculates the increase from the former proportion to the latter proportion, and uses the resulting increase as an estimate of the predicate’s suspiciousness. *SOBER* [13] introduces the concept of statement evaluation bias to express the probability that a predicate is evaluated to be true in an execution. By collecting such evaluation biases of a statement in all failed executions and those in all passed executions, *SOBER* compares the two distributions of evaluation biases, and accordingly estimates the suspiciousness of the predicate [13].

Recent work has improved the effectiveness and efficiency of predicate-level statistical debugging. *Adaptive Bug Isolation* showed that the number of program points that need to be instrumented and monitored to identify fault predicting predicates can be significantly reduced with adaptive sampling [14]. The reduction in the number of instrumented program points improves overall efficiency of predicate-level approaches but not the effectiveness. *Compound boolean predicates* combine existing *CBI* predicates together via boolean expressions. The result is improved effectiveness but increased overhead [15]. In their work with the *Holmes* debugging tool, Chilimbi et al. explored profiling paths instead of predicates to identify sources of program

failures [16]. Profiling paths, as opposed to predicates, requires additional instrumentation but can provide more context on how faults are exercised which can aid developers in debugging. Finally, Zhang et al. showed that *Boolean expression short-circuit rules* used in some compilers can cause the effectiveness of predicate-based approaches to vary significantly for some subject programs [17].

While there has been significant progress in the field of predicate-level statistical debugging there has been little work on improving predicates themselves. In each of the previously described tools the predicates are determined before runtime and are the same at each instrumented program point. The *elastic predicates* presented in Chapter 3 offer a paradigm shifting change to predicate-level statistical debugging. Instead of employing static predicates, the program is profiled to compute summary statistics such as the mean and standard deviation for each instrumented program point. The results are *elastic predicates* which adapt to instrumented program points during execution.

Furthermore, the predicate suspiciousness estimates employed in previous approaches suffer from confounding bias. While the confounding bias in the predicate suspiciousness estimates is related to the bias identified in statement suspiciousness estimates by Baah et al., working with predicates, as opposed to statements, creates different challenges [31, 32, 33, 34]. Predicates encode significantly more program state than statements, resulting in different definitions of control and data flow dependencies. Furthermore, the additional state encoded in predicates enables additional confounding biases to be identified and reduced or eliminated. Our work on reducing confounding biases in predicate suspiciousness estimates is presented in Chapter 4. It significantly improves the effectiveness of each of the approaches presented in this section.

2.2 State-altering Approaches

State-altering approaches to fault localization attempt to identify the cause of program failure by repeatedly altering program states and re-executing the program. The *Delta Debugging* framework systematically narrows down the variables and values relevant to the failure, by iteratively modifying execution state for each test case and observing if there is any difference in the outcome [18, 19]. The process identifies cause-effect chains relevant to a failure and these chains lead to the isolation of the faulty statement [20]. Recently, the efficiency of *Delta Debugging* has been significantly improved by encoding test case inputs into a hierarchy [21]. *Interesting Value Map Pairs (IVMP)* is a similar but more aggressive state-altering approach than *Delta Debugging*. *IVMP* potentially modifies values at every executed statement in a failing test case. This improves effectiveness but leads to a less efficient approach [22, 23].

Predicate Switching [24] differs from both *Delta Debugging* and *IVMP* by only altering the outcomes of branches during the execution of a failing run. As a result, *Predicate Switching* only provides a subset of the

state alterations that are performed in *IVMP* and *Delta Debugging*. Moreover, because predicate switching only alters control flow it may cause the execution of the subject program to enter an inconsistent program state [24].

In general state-altering approaches can be time consuming and require an oracle to determine the success or failure of each altered program. Often, providing such an oracle is infeasible in practice [31]. Furthermore studies suggest that stochastic distributions within subject programs can significantly degrade the effectiveness of state-altering analysis [26].

2.3 Program Slicing Approaches

A *program slice* consists of the parts of a program that can affect a specified point of interest. The specified point of interest is known as the *slicing criterion*. The *slicing criterion* is specified by a line number in the program and a variable name. The task of computing the parts of the program that can affect the slicing criterion is called *program slicing*. The original application of *program slicing* was fault localization. The *slicing criterion* reflected an incorrect value and program slicing was employed to narrow the search for the faulty statement(s) causing the incorrect value [35]. Since its inception, various different notions of program slices have been proposed, as well as a number of approaches for relating program slicing to fault localization. Here we review the difference between static and dynamic program slices and discuss their relation to slicing-based fault localization approaches.

2.3.1 Static Program Slicing

Static program slicing identifies the subset of program statements that may affect the value of a variable at a particular program point, without making any assumptions about program input. Static slices can be obtained by computing consecutive sets of transitively relevant statements, according to data and control flow dependences [35, 36]. An alternative method for computing static slices restates the problem of static slicing in terms of a reachability problem in a program dependence graph [37, 38].

In each of these slicing methods statements and control predicates within the slice are gathered by a backward traversal of the program's control flow graph or program dependence graph, beginning at the *slicing criterion*. Thus these slices are referred to as *backward static slices*. However, a *forward static slice* is also possible [39]. A *forward slice* consists of all statements and control predicates dependent on the slicing criterion. Statement dependence is defined as *the values computed at that statement that depend on the values computed at the slicing criterion or the values computed at the slicing criterion that determine if the statement under consideration is executed* [39]. Backward and forward slices are computed in a similar manner but

forward slices require tracing dependences in the forward direction. However, regardless of the static slicing method the goal is the same: to narrow the search for statements which may affect the slicing criterion.

2.3.2 Dynamic Slicing

In *dynamic program slicing*, only the dependences that occur in a specific execution of the program are taken into account. A *dynamic slicing criterion* specifies the input, and distinguishes between different occurrences of a statement in the execution history. The difference between static and dynamic slicing is that dynamic slicing assumes fixed input for a program, whereas static slicing does not make assumptions regarding the input [40, 41, 42]. The result is that the statements included in a dynamic slice are a subset of the statements included in a static slice. In the context of fault localization, dynamic slices further narrow the search for the faulty statement causing an observed program failure.

A number of hybrid approaches employing a combination of static and dynamic information to improve the efficiency of program slice computation have been proposed. Kamkar uses static information in order to decrease the number of computations that have to be performed at run-time for dynamic slicing [43]. Venkatesh, Ning et al., and Field et al. consider situations where only a subset of the inputs to the program are constrained [44, 45, 46]. More recent work has shown that efficient fully dynamic program slice computation is possible without abstractions or static analysis [47, 48].

Many approaches employ dynamic program slicing to assist in fault localization. Demillio and Pan combine dynamic slicing with mutation testing in a process called critical slicing to further narrow the search for program statements that are relevant to a fault [49]. Agrawal et. al. combine dynamic slicing with automated backtracking in *SPYDER* in an attempt to model the subtasks a programmer goes through in the process of localizing a fault [50, 51]. Fritzson et al. combine functional testing and dynamic slicing to create the interactive, semi-automated fault localization approach *algorithmic debugging* [52].

Furthermore, to help narrow down the faulty statements in a dynamic slice, ranking techniques to place statements in descending order of suspiciousness have been proposed [53, 54, 55]. While these rankings are useful, they have not been shown to be more effective than the leading statistical or state-altering fault localization approaches [55].

Overall, slicing-based fault localization approaches report a subset of program statements where the subset can be large. Unless techniques are used to rank statements in the computed slices, a developer may have to look through many statements before the fault can be found. In contrast, state-altering and statistical approaches compute a ranked list of program statements in which a faulty statement is very likely to be

given a high rank. This difference makes it difficult to compare slicing-based fault localization approaches to statistical and state-altering fault localization approaches.

2.4 Other Approaches to Fault Localization

Several fault localization approaches cannot be categorized as statistical, state-altering or slicing-based. These approaches have not been shown to be as effective as the previously described tools, however they have influenced the design of the most effective fault localization techniques and are useful in preventing errors.

Invariant-based approaches formulate program invariants regarding the proper behavior of a subject program. When the formulated invariants are violated the behavior is reported to the developer as a possible error. The *Daikon* tool automatically infers likely program invariants by dynamically analyzing program executions [56]. The inferred invariants are inserted into the program and report potential errors when violated at runtime.

DIDUCE also infers likely invariants [57]. Again, the most restrictive invariants are initially hypothesized, however in *DIDUCE* as invariant violations are detected and acquitted of errors, the hypothesized invariant is relaxed to allow for newly-encountered program behavior. In *AccMon*, memory location access invariants are identified [58]. When an instruction accesses the restricted memory locations it is suspected to be a memory corruption error and is reported.

Check 'n' Crash derives error conditions in a program statically and generates test cases to dynamically determine if the error exists [59]. *Eclat* infers an operational model of the correct behavior of a program and identifies inputs whose execution patterns differ from the model. Those inputs with a differing execution pattern are likely to reveal errors [60]. Similarly, the *FindBugs* tool automatically detects a commonly observed set of predefined error patterns in Java programs [61]. The *PathExpander* tool provides support to increase the path coverage of dynamic error-detection tools by executing non-taken paths in a sandbox environment [62]. This allows for error detection in paths that would have otherwise not been taken and analyzed.

Most fault localization tools identify faulty statements causing an error. However, the goal of the analysis for some tools is to facilitate developers in explaining *how* an identified fault causes an error. The work of Groce et. al., uses distance metrics to compare passing and failing program executions and isolate the differences between them [63, 64]. The identified differences are used to shed light on how a fault causes an error. Ko and Myers developed a debugging tool called *The Whyline* to help developers better understand program behavior [65, 66]. *The Whyline* allows developers to select a question concerning the output of

a program, and the tool then uses a combination of static and dynamic analysis techniques to search for possible explanations.

2.5 Failure Specific Approaches

There are a number of fault localization approaches that focus only on localizing faults resulting in specific types of failures. Here we review each of these approaches and describe the types of failures they target. In general, state-altering, slicing-based and statistical approaches to fault localization, such as the one presented in this dissertation, target a larger set of subject programs and potential failures than failure specific approaches.

Valgrind and *Purify* are fault localization tools used specifically to help identify faulty statements that cause buffer overflows and memory leaks [67, 68]. *CCured* is a technique for verifying the type-safety of pointers both statically and during runtime, which can also be used to find faulty statements leading to potential memory corruption [69]. However, *CCured* requires modifications to the original subject program source code which most other fault localization approaches do not. *HeapMon* takes advantage of extra cores in hardware to improve the efficiency of error monitoring for heap memory corruption [70]. Similarly, *SafeMem* exploits *error correcting code* memory technology to detect memory leaks and memory corruption [71].

Security attacks present the most opportunities for failure specific tools. One of the most common security attacks is a buffer overflow attack. As a result several tools to identify statements leading to buffer overflow failures exist. *Write Integrity Testing (WIT)* uses a combination of static analysis and runtime instrumentation to ensure that instructions do not write to unintended storage locations, and control does not transfer to unintended targets [72]. The average space and runtime overhead of *WIT* is 10%. Ruwase and Lam’s *CRED* tool performs bounds-checking in order to detect buffer overflows [73]. For some subject programs *CRED* can incur less overhead than *WIT* (down to 1%) but for other subject programs the overhead drastically rises to 130%.

Other security attacks present opportunities as well. *Pointer taint analysis* can be used to detect security attacks that can be associated with pointer-related failures [74]. Pointers within the subject programs are considered tainted if user input can be used to compute the pointer value. Thus, whenever a tainted value is dereferenced during execution, a security attack is detected. *CP-Miner* is a technique that searches for copy-paste errors in large-scale software systems, while *EXPLODE* focuses on identifying data integrity issues in storage systems [75, 76].

2.6 Static Approaches

While the previous approaches described in this section are dynamic approaches that require program execution, there has also been significant work on static approaches for localizing faults. The *LCLint* checking tool assumes annotations are written in a program to explicitly describe the results of functions and the values of parameters and global variables. These annotations are exploited by the tool to detect faults at compile time [77]. The *PREfix* tool performs compile-time analysis of a program to symbolically execute functions while modeling memory and identifying any observed inconsistencies that may be related to a fault [78].

Jackson and Vaziri developed an approach in which a procedure is translated into a relational formula, which is then joined with the negation of the procedure’s specification [79]. A constraint solver is then used to search for potential executions of the code that can violate the procedure specification. Any violations are likely to be associated with faulty statements. The *SLAM* toolkit creates abstractions of C code using iterative refinement, based on a specified temporal safety property of interest [80]. The specified property is automatically validated using the tool. *BLAST* also verifies safety properties of C programs [81].

CQual assumes that programs written in languages with static type systems are annotated with type qualifiers [82, 83]. Automatic type qualifier inference is used to determine any remaining qualifiers, and then the system checks the annotations for consistency. The *PSE* technique takes as input the type and source code location of an execution failure and tracks the relevant value of interest back from the point of the failure through the points in the program where that value may have originated [84]. Xie and Aiken present a failure-detection tool that exploits advances in boolean satisfiability solvers to translate programs into boolean formulas that can be solved to check for violations that can be correlated with failures [85]. The *Extended Static Checker* for Java looks for common faults at compile-time by way of an annotation language in which a developer formally expresses design decisions [86]. It has been observed that static techniques for localizing faults often require programmers to modify software with annotations or specifications, and this can potentially place a significant burden on developers. To address this issue, a body of research has been conducted on using data mining techniques to automatically infer specifications from software [87, 88, 89, 90].

2.7 Casual Inference

This section presents background information needed to understand how causal inference is applied to the fault localization approach described in this dissertation. Section 2.7.1 presents an overview of causal graphs and models and Section 2.7.2 shows how causal graphs and models can be combined to estimate causal effects.

2.7.1 Causal Graphs and Models

The fault localization approach in this dissertation builds on the *causal graphs* presented in Pearl's structural model [91]. In Pearl's model, causal graphs are used to represent: (1) the causal assumptions that permit statistical techniques to be used with observational data to make inferences about causality, and (2) changes such as treatments. A *causal graph* or *causal Bayesian network* is a directed acyclic graph G whose nodes represent random variables corresponding to causes and effects and whose edges represent causal relationships.

An edge $X \rightarrow Y$ indicates that X (potentially) causes Y . Each random variable X has a probability distribution $P(x)$, whose form may or may not be known. All causal effects associated with the *causal model* $M = (G, P)$ are *identifiable*, which means the effects can be estimated, if M is *Markovian*. Here, *Markovian* means that each random variable X_i is conditionally independent of all of its nondescendants, given the values of its parents (immediate predecessors) PA_i in G [92]. If M is Markovian then the joint distribution of the random variables can be factored as shown in Equation 2.1. In Equation 2.1 x_i and pa_i reflect outcomes of the random variables X_i and PA_i respectively. This established notation is used throughout this section to distinguish random variables from their outcomes [93].

$$P(x_1, x_2, \dots, x_n) = \prod_i P(x_i | pa_i) \quad (2.1)$$

Pearl and Verma [91] proved that M will satisfy this *parental Markov condition* if it corresponds to a *functional causal model*, such that, for each node $X(i)$ in G , the relationship between $X(i)$ and its parents can be described by a *structural equation* of the form shown in Equation 2.2.

$$x_i = f_i(pa_i, u_i) \quad (2.2)$$

Here f_i , which represents a causal process, may be any function, and for $i = 1, \dots, n$ the random variables U_i , which represent random errors due to unobserved variables, are independent of each other. If X_i has no parents then $x_i = f_i(u_i)$. M is Markovian if it represents functional relationships among a set of random variables and if any external sources of error are mutually independent. It is important to note that, for the purposes of causal inference, the functions f_i and the distributions of the error variables U_i need not be known. Thus, M is nonparametric and U_i does not need to be explicitly represented in G .

2.7.2 Causal Estimation

This model of causality has been influential in a number of fields, including economics, epidemiology, and social science [94]. For a binary cause, each member of the study population can be exposed to two states:

treatment and *control*. A *treatment* is an intervention an investigator applies to a set of members to assess its effects relative to no intervention, the *control*. The states *treatment* and *control* correspond to the values of a treatment variable T . For treated units $T = 1$, for control units $T = 0$. Given a measurable outcome variable Y over the study population, there are two potential outcome random variables: Y^1 and Y^0 .

For a unit i , the potential outcome under the treatment state is y_i^1 , and the potential outcome under the control state is y_i^0 . The causal effect of the treatment for unit i is $\tau_i = y_i^1 - y_i^0$. However, it is not usually possible to calculate causal effects for individual members. As a result the average causal effects for the population are estimated instead. Equation 2.3 shows how the average treatment effect in the population is calculated [91]. Within Equation 2.3 $E[\cdot]$ denotes the expectation operator.

$$\tau = E[Y^1] - E[Y^0] \quad (2.3)$$

Many real-world problems, such as evaluating a medical treatment, call for estimating the average treatment effect τ in a population from a sample S of n members. Here, S_1 is the subset of S consisting of the treated members, and S_0 is the subset of S consisting of the control members. The natural estimator of τ is the difference of the sample means of the outcome values for those in the treatment group and those in the control group [94]. This estimate $\hat{\tau}$ is shown in Equation 2.4.

$$\hat{\tau}_{re} = \frac{1}{S_1} \sum_{i \in S_1} y_i - \frac{1}{S_0} \sum_{i \in S_0} y_i \quad (2.4)$$

In an ideal randomized experiment, units would be assigned to the treatment group or the control group randomly. Such an assignment implies that the treatment indicator variable T is independent of the potential outcomes Y_1 and Y_0 . In this case, it can be shown that $\hat{\tau}_{re}$ is an unbiased and consistent estimator of the average treatment effect τ meaning that $E[\hat{\tau}_{re}] = \tau$ [94].

However, in an observational study treatment effects are not under the control of the investigator, often because the data being analyzed occurred in the past. In general, treatment selection is not random and hence the treatment indicator variable T is not independent of the potential outcomes Y_1 and Y_0 . Under these circumstances, the estimator $\hat{\tau}_{re}$ is likely to be biased such that $E[\hat{\tau}_{re}] \neq \tau$ [94].

For an observational study, a better estimator of the average treatment effect τ is needed. Although the process of treatment selection is seldom fully understood, it is often possible to characterize it in terms of one or more variables that are known or suspected of playing an important role in the selection. For example, a physician would consider the symptoms, vital signs, and medical history of a patient before selecting a treatment. These variables are *covariates* of the treatment indicator T .

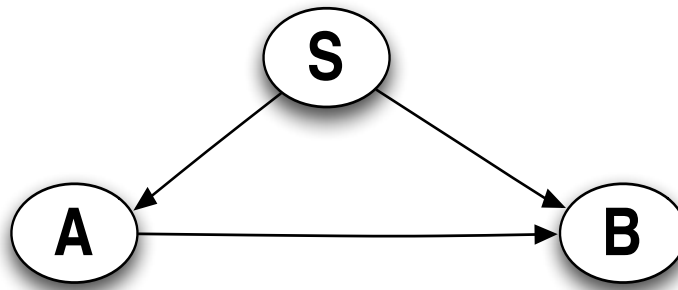


Figure 2.1: Causal graph illustrating *Back-Door Criterion*.

If a set X of covariates accounts well for which individuals or units received the treatment and which did not, then it is possible to reduce or eliminate selection bias when estimating the average treatment effect τ on an outcome Y , by conditioning on or adjusting for X [94]. Pearl's *Back-Door Criterion* [92] defines, in terms of causal graphs, the characteristics that make a set X of covariates suitable for this purpose. Before presenting the *Back-Door Criterion*, the concepts of *blocking* and *d-separation* are defined [92]. Note that, as opposed to typical directed graph terminology, this definition refers to causal graph *paths* where the edge may be directed either forward or backward along a path.

Definition 1. A set S of nodes in a causal graph G is said to **block** a path p if either: (1) p contains a chain $U \rightarrow M \rightarrow V$ or a **fork** $U \leftarrow M \leftarrow V$ whose middle node M is in S , or (2) p contains at least one **collider** $U \rightarrow M \leftarrow V$ such that the middle node M is not in S and no descendant of M is in S . If S blocks all paths from A to B , it is said to **d-separate** A and B .

Definition 2. A set S of nodes satisfies the **Back-Door Criterion** relative to a pair of nodes A, B in a causal graph G if:

1. No node in S is a descendant of A .
2. S blocks every path between A and B that contains an edge into A .

Similarly, if A and B are two disjoint subsets of nodes in G , then S is said to satisfy the **Back-Door Criterion** relative to A, B if it satisfies the criterion relative to any pair $A_i \in A, B_j \in B$.

The name of the *Back-Door Criterion* comes from condition (2), which requires that the paths entering A through the *back door* be blocked. Figure ?? shows an example of a causal graph, in which there is a *back-door* path from A to B through S [94, 92]. The *Back-Door Criterion* unifies a number of strategies for how to solve treatment selection bias [94], including conditioning, stratification, and matching. If X is a set of variables that blocks all back door paths in a causal graph between a treatment indicator T and an outcome

variable Y , then those paths do not contribute to the estimation of the effects between T and Y [94]. Hence, by conditioning on X , the average treatment effect of T upon Y can be estimated without confounding bias.

There are a number of approaches to using a set X of covariates that satisfy the Back-Door Criterion, relative to a treatment indicator T and an outcome variable Y , to construct an unbiased and consistent estimator of the average treatment effect τ [94]. In this dissertation, a standard approach based on linear regression is employed [95]. A linear regression model for estimating τ is shown in Equation 2.5.

$$Y = \alpha + \tau T + \beta X + \epsilon \quad (2.5)$$

In this linear regression model, α is an intercept, τ and β are coefficients of T and X respectively, and ϵ is an error term that is uncorrelated with T . The least-squares estimate of τ , which we denote by $\hat{\tau}_p$, is unbiased and consistent [94].

2.8 Summary

A myriad of approaches to localizing sources of faults exist in the software engineering community. Statistical debuggers measure and rank the likelihood that each program statement (or predicate) reflects the fault. State-altering approaches create similar rankings but rely on repeated program state modifications and program executions, frequently resulting in inefficient analysis. Program slicers identify those program statements that can affect a faulty outcome, but do not rank them. Static approaches are employed to prevent faults as well but their inability to execute a faulty subject program restricts their fault localization effectiveness. A number of other approaches tailored to specific faults exist but they are ineffective and inapplicable to large classes of faulty programs.

However, no fault localization approach is well suited to an important class of software, including simulations and computational models, which employ floating-point computations and continuous stochastic distributions to represent, or support the evaluation of, an underlying model. This dissertation addresses this deficiency with three contributions to improve the effectiveness of predicate-level statistical debuggers. These contributions are presented in the three chapters which follow: Chapter 3: *Elastic Predicates*, Chapter 4: *Reducing Confounding Bias* and Chapter 5: *Fuzzy Passing Extents*. The contributions related to reducing confounding bias in Chapter 4 make extensive use of causal inference which is reviewed here in Section 2.7. The remainder of this dissertation presents and evaluates each of our contributions.

Chapter 3

Elastic Predicates

Automated fault localization using statistical debugging methods has enabled partial automation of a typically manual and laborious task. Up until now statistical debugging methods have assumed discrete data types and static predicates, which has supported advancements in related research, but has precluded practical consideration of important classes of programs. In particular, statistical debuggers have not been well suited to simulations and computational models, which employ floating-point computations and continuous stochastic distributions to represent, or support evaluation of, an underlying model. The limitations of traditional statistical debugging methods present an opportunity to improve the effectiveness of the fault localization capabilities for predicate-level statistical debuggers. The improvements are applicable to general purpose software and particularly simulations and computational models.

Predicate-level statistical debuggers represent a class of fault localization approaches that share a common structure. Each approach consists of a set of conditional propositions, or predicates, tested at various points in a faulty subject program. The likelihood or *suspiciousness* that each predicate p reflects the fault is estimated based on how frequently predicate p is true in successful (passing) and failing test cases for a subject program [9, 10]. The predicates in the subject program are ranked, based on their estimated suspiciousness, and the rankings are provided to developers to assist in finding the fault.

Predicates in existing statistical debuggers are *static* in the sense that the conditional propositions tested in the debugger are specified before the subject program is executed. For example, in the canonical predicate-level statistical debugger, Cooperative Bug Isolation (CBI), three static predicates are tested for each assignment to a variable x_y in a statement y (denoted x_y). These predicates are: $(x_y > 0)$, $(x_y = 0)$ and $(x_y < 0)$ [9, 10, 11]. The decision to compare the value of x_y to 0 in each of these predicates is made before the subject program is executed, making these predicates *static*.

While predicate-level statistical debuggers employing static predicates have been shown to be effective for a variety of general-purpose software applications, they can be improved. Elastic predicates offer such an improvement.

Unlike their static counterparts, *elastic predicates*, as we define them here, adapt to profiled variable values to create predicates that better localize faults. Given a measure to estimate predicate suspiciousness, a *maximized elastic predicate* identifies the values assigned to a variable x in a statement y (denoted x_y) that maximize the suspiciousness of the predicate. These *maximized elastic predicates* are the most effective predicates for fault localization because each predicate is constructed to identify the most suspicious range of values assigned to each x_y .

Unfortunately, generating *maximized elastic predicates* is impractical. *Suspiciousness maximized elastic predicates* require all of the values assigned to each x_y to be stored and sorted. For many subject programs, the space required to store all of the values assigned to each x_y will exceed the space available on a modern workstation. Furthermore, for most subject programs, sorting the assigned values to each x_y cannot be performed quickly enough to make the resulting *maximized elastic predicate* useful.

However, other *elastic predicates* that can be computed efficiently in terms of time and space are realizable. The *elastic predicates* presented in this section use summary statistics such as the mean, μ_{x_y} , and standard deviation, σ_{x_y} , of the values assigned to a variable x_y to cluster together similar values.

The use of summary statistics to compute elastic predicates is not an arbitrary choice. Summary statistics enable the resulting *elastic predicates* to be computed in an online manner, which avoids any need to store each value assigned to each x_y [96]. Furthermore, because the summary statistics capture the dispersion of variable values without requiring sorting or optimization routines, the predicates can be computed efficiently. While *elastic predicates* based on summary statistics do not maximize the suspiciousness estimate at each instrumented program point, the results presented in Section 3.4 show they approach the improvements in effectiveness that are possible with *maximized elastic predicates* even in the face of sparse sampling and incomplete test suites.

3.1 Motivating Example

An example helps elucidate the shortcomings of static predicates and the improvements that are possible by using *elastic predicates* which adapt to variable values profiled during program execution. Figure 3.1 shows the source code of the `more_arrays()` function in the 1.06 version of the GNU implementation of BC, a basic command-line calculator tool [97]. The `more_arrays()` function is responsible for increasing the number of arrays needed for BC computing. The logic within the function is an example of buffer reallocation. Statement

```

152 void
153 more_arrays()
154 {
155     int indx;
156     int old_count;
157     bc_var_array **old_ary;
158     char **old_names;
159
160     /* Save the old values. */
161     old_count = a_count;
162     old_ary = arrays;
163     old_names = a_names;
164
165     /* Increment by a fixed amount and allocate. */
166     a_count += STORE_INCR;
167     arrays = (bc_var_array **) bc_malloc (a_count*sizeof(bc_var..
168     a_names = (char **) bc_malloc (a_count*sizeof(char *));
169
170     /* Copy the old arrays. */
171     for (indx =1; indx < old_count; indx++)
172         arrays[indx] = old_ary[indx];
173
174
175     /* Initialize the new elements. */
176     for (; indx < v_count; indx++)
177         arrays[indx] = NULL;
178
179     /* Free the old elements. */
180     if (old_count != 0)
181     {
182         free (old_ary);
183         free (old_names);
184     }
185 }

```

Figure 3.1: The source code for the `more_arrays()` function in BC.

167 allocates a larger chunk of memory. Statement 171 is the top of a loop that copies values over from the old, smaller array, `old_ary`. Statement 176 is the top of a loop that completes the resize by zeroing out the new extra space in `arrays[indx]`. However, there is a fault in the function. The allocation in Statement 167 requests space for `a_count` items. The copying loop in Statement 171 ranges from 1 through `old_count - 1`. The zeroing loop in Statement 176 continues from `old_count` through `v_count - 1`. Here lies the fault: the new storage buffer has room for `a_count` elements, but the second loop is incorrectly bound by `v_count`. Thus, when `v_count` is greater than `a_count` the program fails.

Recall, in CBI, three static predicates partition the values assigned to each variable x in a statement y : $(x_y > 0)$, $(x_y = 0)$ and $(x_y < 0)$. Unfortunately, these predicates do not effectively localize the fault in `more_arrays()`. The predicates are chosen before the program is executed and are unable to adapt when almost all of the values assigned to each variable in each statement of `more_arrays()` are greater than zero and satisfy the same predicate, $(x_y > 0)$.

Section 3.3 demonstrates how *elastic predicates* address this problem by employing predicates which adapt to profiled variable values. However, first, Section 3.2 reviews the different formulas used to estimate the suspiciousness of static and elastic predicates. Then, Section 3.4 presents an experimental study which

evaluates the effectiveness and efficiency of elastic predicates compared to several competing fault localization approaches, including CBI.

3.2 Suspiciousness Estimates

The extent to which a predicate p reflects subject program failure is measured through a *suspiciousness estimate*. Estimating the suspiciousness of a predicate p requires two data structures for each executed test case: a Boolean variable R and a Boolean vector V . The Boolean variable R indicates if the test case succeeded or failed. Each entry within the Boolean vector V indicates if the corresponding predicate is observed to be true during test case execution [9, 10]. The data recorded in the R and V for each test case is aggregated into the following terms to estimate predicate suspiciousness: s is the total number of tests that succeed (pass), f is the total number of tests that fail, s_p is the number of tests that succeed where p is true and f_p is the number of tests that fail where p is true. Different suspiciousness estimates use these terms differently.

3.2.1 Specificity

All suspiciousness estimates include an estimate of the probability of subject program Q failing given that predicate p is true, $\Pr(Q \text{ fails} \mid p=\text{true})$. The simplest estimate of $\Pr(Q \text{ fails} \mid p=\text{true})$ is the ratio of failing test cases where predicate p is true to total test cases where predicate p is true. This ratio, shown in Equation 3.1, is known as the *specificity* measure.

$$specificity = \frac{f_p}{f_p + s_p} \quad (3.1)$$

3.2.2 Tarantula

The *Tarantula* suspiciousness measure, shown in Equation 3.2, is closely related to the *specificity* measure.

$$Tarantula = \frac{\frac{f_p}{f}}{\frac{f_p}{f} + \frac{s_p}{s}} \quad (3.2)$$

Tarantula differs from *specificity* because it includes the number of failed test cases, f , in the numerator and denominator and it includes the number of successful test cases, s , in the denominator. However, when $s = f$

as shown in Equation 3.3, *Tarantula* reduces to *specificity*.

$$Tarantula = \frac{\frac{f_p}{f}}{\frac{f_p}{f} + \frac{s_p}{s}} = \frac{\frac{f_p}{f}}{\frac{f_p}{f} + \frac{s_p}{f}} = \frac{\frac{f_p}{f}}{\frac{f_p + s_p}{f}} = \frac{f_p}{f_p + s_p}, \text{ when } s = f \quad (3.3)$$

3.2.3 F_1 Measure

In information retrieval, the F_1 measure balances an estimate of $\Pr(Q \text{ fails} \mid p=\text{true})$ with an estimate of the probability that predicate p is true given that the subject program Q failed, $\Pr(p = \text{true} \mid Q \text{ fails})$. The *sensitivity* measure, shown in Equation 3.4, is used in the F_1 measure to estimate $\Pr(p = \text{true} \mid Q \text{ fails})$.

$$sensitivity = \frac{f_p}{f} \quad (3.4)$$

The inclusion of *sensitivity* within a suspiciousness estimate addresses issues that occur when a predicate p is true in very few failing test cases. Without *sensitivity*, if there are many successful (passing) test cases where p is true, the overall sample of test cases where p is true will be unbalanced and the *specificity* will be low even if there are very few failed test cases. Conversely, if there are few successful (passing) test cases where p is true, the overall sample of test cases where p is true will be small and the *specificity* could be high even if there are many failed test cases where predicate p is not true. By balancing *specificity* and *sensitivity* via their harmonic mean, the F_1 Measure, these issues are addressed. The F_1 Measure is shown in Equation 3.5.

$$F_1 Measure = \frac{2}{\frac{1}{\frac{f_p}{f}} + \frac{1}{\frac{f_p}{f_p + s_p}}} \quad (3.5)$$

3.2.4 Importance

Liblit's *Importance* measure is closely related to the F_1 measure [9, 10, 11]. However, along with accounting for s_p and f_p , *Importance* also accounts for $s_{p \text{ obs}}$ and $f_{p \text{ obs}}$. The terms $s_{p \text{ obs}}$ and $f_{p \text{ obs}}$ are the number of test cases that succeed or fail when p is evaluated (true or false), respectively.

Within the *Importance* measure, both $s_{p \text{ obs}}$ and $f_{p \text{ obs}}$ are used to define *Increase*, shown in Equation 3.6.

$$Increase = \frac{f_p}{f_p + s_p} - \frac{f_{p \text{ obs}}}{f_{p \text{ obs}} + s_{p \text{ obs}}} \quad (3.6)$$

Once the fault in a subject program has been triggered, the subsequent predicates that are tested are more likely to appear in failing executions. This situation creates bias in the *specificity* measure. The first term in *Increase* is identical to the *specificity* measure. However, the second term in *Increase* is meant to factor out the bias by ensuring that predicate p is scored, not by the chance that p implies failure, but by

how much difference it makes that p is true versus simply reaching the statement where p is evaluated (true or false) [10, 11].

The *Importance* measure, shown in Equation 3.7, mirrors the F_1 Measure, except that it replaces the *specificity* measure with *Increase*.

$$Importance = \frac{2}{\frac{1}{f_p} + \frac{1}{Increase}} \quad (3.7)$$

3.2.5 Ochiai

Other statistics besides the harmonic mean can be employed to balance the measures *specificity* and *sensitivity*. The *Ochiai* suspiciousness estimate, shown in Equation 3.8, employs the geometric mean to balance *specificity* and *sensitivity*.

$$Ochiai = \sqrt{\frac{f_p}{f} \times \frac{f_p}{f_p + s_p}} \quad (3.8)$$

Both the geometric mean (*Ochiai*) and harmonic mean (F_1) penalize predicates with uneven measures of *specificity* and *sensitivity*. However, the harmonic mean penalizes uneven performances more harshly than the geometric mean. Thus, given a list of ranked predicates, unbalanced predicates (high specificity and low sensitivity or low specificity and high sensitivity) are more likely to be close to the top of the list when the *Ochiai* measure is used to estimate suspiciousness, as opposed to the F_1 measure.

3.3 Methodology

Numerous approaches have been proposed to identify predicates that are good failure predictors [9, 10, 13, 11]. The most effective of these approaches analyze variable values within a program. In this section two elastic predicate instrumentation schemes which employ summary statistics such as the mean and standard deviation to analyze variable values are presented. In Section 3.3.1 the elastic *single variable* instrumentation scheme uses the mean, μ_{x_y} , and standard deviation, σ_{x_y} , of the values assigned to a variable x in a statement y to cluster together similar values. The scalar pairs instrumentation scheme considers the difference between the value of x_y and another in scope similarly typed variable q in statement i to capture the relationships among multiple variables which cannot be detected by the single variable instrumentation scheme. The elastic *scalar pairs* instrumentation scheme presented in Section 3.3.2, uses the mean $\mu_{x_y - q_i}$ and standard deviation $\sigma_{x_y - q_i}$ of the difference between the value of x_y and q_i to create partitions that cluster together differences which are similar. Section 3.3.3 describes the implementation of the instrumentation mechanism for both of these schemes and Section 3.3.4 presents the predicates that can be inferred from both of the instrumentation schemes.

Table 3.1: Fundamental single variable elastic predicates.

$x_y > (\mu_{x_y} + 3\sigma_{x_y})$
$(\mu_{x_y} + 3\sigma_{x_y}) \geq x_y > (\mu_{x_y} + 2\sigma_{x_y})$
$(\mu_{x_y} + 2\sigma_{x_y}) \geq x_y > (\mu_{x_y} + \sigma_{x_y})$
$(\mu_{x_y} + \sigma_{x_y}) \geq x_y > \mu_{x_y}$
$\mu_{x_y} = x_y$
$(\mu_{x_y} - \sigma_{x_y}) \leq x_y < \mu_{x_y}$
$(\mu_{x_y} - 2\sigma_{x_y}) \leq x_y < (\mu_{x_y} - \sigma_{x_y})$
$(\mu_{x_y} - 3\sigma_{x_y}) \leq x_y < (\mu_{x_y} - 2\sigma_{x_y})$
$x_y < (\mu_{x_y} - 3\sigma_{x_y})$

3.3.1 Single Variable

The *single variable* instrumentation scheme partitions the set of possible values that can be assigned to a variable x in a statement y . In CBI, three familiar, static predicates are employed to partition the values for each x_y : $(x_y > 0)$, $(x_y = 0)$ and $(x_y < 0)$. Recall, these three predicates are static because the decision to compare the value of x_y to 0 in each of these predicates is made before subject program execution. In contrast, the nine elastic predicates presented in Table 3.1 use summary statistics of the values assigned to x_y to create partitions that cluster together values which are a similar distance and direction from μ_{x_y} . The program that performs the instrumentation to construct the elastic *single variable* predicates in Table 3.1 is described in detail in Section 3.3.3.

The decision to use nine elastic predicates to partition the values assigned to variable x in statement y for three standard deviations (σ_{x_y}) above and below the mean (μ_{x_y}) is deliberate. This partitioning has been effective to capture the normal dispersion of data for many problems in numerous domains [95, 98, 99, 100, 101, 102]. Furthermore, computer graphics researchers have had success improving the efficiency of shaders by replacing functional expressions with summary statistics over the input domain [103]. In Section 3.4.9 a reduced set of these elastic *single variable* predicates is explored that provides most of the effectiveness of the nine predicates shown in Table 3.1 while requiring less subject program execution time.

The faulty program BC in Figure 3.1 demonstrates the utility of the *single variable* elastic predicates defined in Table 3.1. Recall, the fault in the `more_arrays` function of BC is difficult to localize with the three static *single variable* predicates because almost all of the values assigned to each variable satisfy the predicate $(x_y > 0)$. However, the elastic predicates in Table 3.1 avoid this problem. Profiling the values assigned to each x_y during execution creates partitions that ensure only similar variable values, in terms of distance and direction from the mean, satisfy the same predicate.

The most suspicious static predicate and the most suspicious elastic predicate for the `more_arrays` function of BC are shown in Table 3.2. The predicates and their suspiciousness estimates are computed using 4,000 randomly generated, valid BC programs as test cases. Table 3.2 shows that the elastic predicate ($indx_{176}$

Table 3.2: The top ranked predicates for Figure 3.1.

Filename	Elastic/CBI	Function	Predicate
storage.c	Elastic	more_arrays()	$indx_{176} > (\mu_{indx_{176}} + 3\sigma_{indx_{176}})$
storage.c	CBI	more_arrays()	$a_count_{166} > 0$

$> (\mu_{indx_{176}} + 3\sigma_{indx_{176}})$ clusters together unusually large values assigned to the variable `indx` in statement 176 and captures the fault. The predicate suggests that failures frequently occur when the input to BC defines an unusually large number of arrays. Specifically, failures occur when BC has room for the number of requested arrays in the input (`a_count`), but the loop is incorrectly bound by a larger number (`v_count`). The location of the fault and the cause of the failure are clear after identification and explanation. However, this fault was present and undiscovered for ten years in BC [97].

In contrast to the top ranked elastic predicate, none of the *single variable* static predicates are particularly suspicious. The most suspicious *single variable* static predicate suggests that some values assigned to the variable `a_count` in statement 166, which are greater than zero, cause the program to fail. While the suggestion is correct it does not lead the developer to the location or the direct cause of the fault. The top ranked elastic predicate does.

This example showcases the ability of elastic predicates to improve the effectiveness of predicate-level statistical debuggers for general purpose software. However, the empirical study presented in the evaluation in Section 3.4 shows that even more significant improvements are possible for software employing floating-point computations and stochastic distributions, including simulations and computational models.

3.3.2 Scalar Pairs

Multiple variables within a program can have important relationships that cannot be captured with a single variable instrumentation scheme. Work on the Daikon project has shown that it is useful to identify and capture the relationships among multiple variables with simple and implicit invariants that aid program evolution and program understanding [56]. Similarly, predicate-level statistical debuggers capture important relationships among multiple variables by identifying invariants that are only violated when the subject program fails. The statistical debugging instrumentation scheme that captures these invariants is the *scalar pairs* instrumentation scheme [10].

In the static *scalar pairs* instrumentation scheme, at each assignment to a variable x in a statement y , all other in-scope, same-typed local or global variables: $q_1, q_2, \dots, q_i, \dots, q_n$ are identified. For each pair of variables the static *scalar pairs* scheme compares the difference of a new value for x_y and the existing value of q_i to zero: $(x - q_i > 0)$, $(x_y - q_i = 0)$, $(x_y - q_i < 0)$.

Table 3.3: Fundamental scalar pairs elastic predicates.

$x_y - q_i > (\mu_{x_y - q_i} + 3\sigma_{x_y - q_i})$
$(\mu_{x_y - q_i} + 3\sigma_{x_y - q_i}) \geq x_y - q_i > (\mu_{x_y - q_i} + 2\sigma_{x_y - q_i})$
$(\mu_{x_y - q_i} + 2\sigma_{x_y - q_i}) \geq x_y - q_i > (\mu_{x_y - q_i} + \sigma_{x_y - q_i})$
$(\mu_{x_y - q_i} + \sigma_{x_y - q_i}) \geq x_y - q_i > \mu_{x_y - q_i}$
$\mu_{x_y - q_i} = x_y - q_i$
$(\mu_{x_y - q_i} - \sigma_{x_y - q_i}) \leq x_y - q_i < \mu_{x_y - q_i}$
$(\mu_{x_y - q_i} - 2\sigma_{x_y - q_i}) \leq x_y - q_i < (\mu_{x_y - q_i} - \sigma_{x_y - q_i})$
$(\mu_{x_y - q_i} - 3\sigma_{x_y - q_i}) \leq x_y - q_i < (\mu_{x_y - q_i} - 2\sigma_{x_y - q_i})$
$x_y - q_i < (\mu_{x_y - q_i} - 3\sigma_{x_y - q_i})$

In contrast, the nine *scalar pair* elastic predicates presented in Table 3.3 use summary statistics of the difference between the new value of x_y and the existing value of q_i to create partitions that cluster together differences which are a similar distance and direction from $\mu_{x_y - q_i}$. The program that performs the instrumentation to construct the elastic *scalar pair* predicates in Table 3.3 is described in detail in Section 3.3.3.

Once again, the use of nine elastic predicates to partition the values of the difference between the new value of x_y and the existing value of q_i is a deliberate decision. The evaluation in Section 3.4.7 shows that these predicates approach the effectiveness of the impractical *maximized elastic predicates* discussed in the chapter introduction. The evaluation also explores a reduced set of elastic *scalar pairs* predicates (three) which provide most of the effectiveness of the nine predicates shown in Table 3.3 while requiring less subject program execution time.

3.3.3 Instrumentation

Employing elastic predicates requires transforming unmodified, subject program source code to native executables with instrumentation that tests conditional propositions based on summary statistics of program points specified by the instrumentation scheme (e.g. single variable or scalar pairs). The process is managed by the instrumenter, **instrument-cc**, whose external behavior mimics that of the native compiler, but internally injects elastic predicate instrumentation into faulty subject programs. From the developer's perspective, the **instrument-cc** command behaves exactly like the **gcc** command with additional command line flags and execution time.

The instrumenter is composed of two different source-to-source transformation routines: **profile-cc** and **elasticate-cc**. The first transformation routine, **profile-cc**, is used to inject instrumentation into the subject program to update a serializable dictionary. The purpose of the dictionary is to track the mean and standard deviation of each specified program point. The injected instrumentation performs the dictionary update using online algorithms requiring only a constant amount of space [96]. Once the subject program is

instrumented by `profile-cc`, it is executed for each provided test case. The test case executions build the dictionary of summary statistics for the specified program points.

After all of the test cases have been executed, the `elasticate-cc` transformation routine injects elastic predicates into the *original unmodified*, subject program. `elasticate-cc` references the dictionary to inject elastic predicates for each specified program point.

3.3.4 Inferred Predicates

The term *fundamental* elastic predicates refers to the *single variable* elastic predicates listed in Table 3.1 and the *scalar pairs* elastic predicates listed in Table 3.3. These are known as *fundamental* predicates because instrumentation is directly injected into the subject program to test if the predicates are true during the execution of a test case. However, *fundamental* predicates can be augmented with additional *inferred* predicates that can be derived offline, after the execution of a test case. An *inferred* predicate is a conditional proposition whose truth can be deduced by computing a logical disjunction of the *fundamental* predicates at a specified program point. For example, given the feedback report from a test case for the BC program in Figure 3.1, it could be inferred that the predicate $indx_{176} \geq \mu_{indx_{176}}$ was true if one of the first five *fundamental single variable* elastic predicates in Table 3.1 was observed to be true for the variable `indx` in Statement 176. Inferred predicates are particularly useful for distinguishing between boundary cases such as $indx_{176} \geq \mu_{indx_{176}}$ versus $indx_{176} > \mu_{indx_{176}}$.

3.4 Evaluation

The utility of a fault localization approach is determined through experimental evaluation. In this section a thorough evaluation of the effectiveness and the efficiency of elastic predicates under a variety of conditions is presented. Sections 3.4.1 and 3.4.2 describe the subject programs and fault localization approaches included in the evaluation. The effectiveness of the approaches is evaluated in Section 3.4.3. Sections 3.4.5 and 3.4.6 explore how effective elastic predicates are in the face of sparse sampling rates and incomplete test suites. In Section 3.4.7 the effectiveness of our elastic predicates, which employ summary statistics, is compared to the effectiveness of the previously described *maximized elastic predicates* for a subset of the subject programs. Then, the efficiency of all of the approaches is presented in Section 3.4.8 and a more efficient, reduced set of elastic predicates is considered in Section 3.4.9. Finally, Section 3.4.10 discusses the validity of the evaluation and Section 3.4.11 discusses the effectiveness of elastic predicates in the face of subject programs with multiple faults.

3.4.1 Subject Programs

Evaluation consists of 28 programs with 328 faulty versions. Programs traditionally used to evaluate fault localization approaches within the Unix suite (`cal`, `col`, `comm`, `spline`, `tr`, and `uniq`), the Siemens suite (`print-tokens`, `print-tokens2`, `replace`, `schedule`, `schedule2`, `tcas`, and `totinfo`) and the programs `sed`, `space`, `flex`, `grep`, `bc` and `gzip` are included in the evaluation. The simulations `bates`, `heston`, `mc`, `euro`, `um-olsr`, `ns2`, `g/g/1`, `m/m/c` and `mmp/d/1` are also included. Each of the simulations is accompanied with actual faults observed in the wild [26].

These subject programs reflect established fault localization benchmarks or widely used simulations employing floating-point computations and continuous stochastic distributions. Table 3.4 shows the characteristics of the subjects. For each subject, the first column gives the program name, the second column provides the ratio of the number of versions used to the number of versions available, the third column gives the number of lines of code for the subject, the fourth column gives the number of test cases and the last column provides a description. For 25 of the faulty versions there were either no syntactic differences between the correct version and the faulty version of the program, or none of the test cases failed when executed on the faulty version of the program. As a result these 25 versions were omitted from the evaluation.

All of the faults included in the subject programs are computation related, as opposed to memory-related. These faults reflect operator and operand mutations, missing and extraneous code, and constant value mutations. For subject program versions with a faulty constant assignment statement, the assignment statement is considered to be examined by a developer when it is directly examined or when a statement explicitly using the constant is examined. Second, for subject program versions where the fault reflects a missing statement, statements directly adjacent to the missing code qualify as the missing statement. These issues are handled the same way in other published research in the fault localization community [7, 12, 22, 23].

3.4.2 Competing Fault Localization Approaches

Four fault localization approaches are included in the evaluation: Cooperative Bug Isolation (CBI), Exploratory Software Predictor (ESP), Tarantula and Interesting Value Map Pairs (IVMP). Here, the different strategies associated with each of the four approaches is summarized.

Cooperative Bug Isolation (CBI)

In the evaluation, CBI is employed with the static *single variable* and static *scalar pairs* instrumentation schemes described in sections 3.3.1 and 3.3.2. It is also employed with the *branches* instrumentation scheme.

Table 3.4: Subject programs used in the evaluation of elastic predicates

Name	# of Vers Used / # of Vers	LoC	# of Tests	Description
cal	19/20	202	162	calender printer
col	29/30	102	156	filter-line reverser
comm	10/12	167	186	file comparer
look	8/14	170	193	word finder
spline	13/13	338	700	curve interpolator
tr	11/11	137	870	character translator
uniq	17/17	143	431	duplicate line remover
print-tokens	5/7	472	4,130	lexical analyzer
print-tokens2	10/10	399	4,115	lexical analyzer
replace	28/32	512	395	pattern replacement
schedule	9/9	292	2,710	priority scheduler
scheule2	9/10	301	2,650	priority scheduler
tcas	41/41	141	1,608	altitude separator
totinfo	23/23	440	1,052	information measure
sed	10/10	6,092	363	stream editing utility
space	30/38	14,382	157	language interpreter
bc	1/1	14,288	4,000	basic calculator
gzip	9/9	7,266	217	compression utility
flex	21/21	10,124	567	lexical parser
grep	17/17	9,089	809	text pattern processor
bates	1/1	8,184	298	options pricing model
heston	1/1	4,095	316	options pricing model
mc euro	1/1	835	242	options pricing model
um-olsr	1/1	14,433	176	network protocol simulator
ns2	1/1	11,258	293	network simulator
g/g/1	1/1	2,247	3,000	queuing simulation
m/m/c	1/1	3,302	3,000	queueing simulation
mmp/d/1	1/1	3,860	3,000	queueing simulation

The *branches* scheme considers each two-way branch statement with two fundamental predicates. The first fundamental predicate asserts that the branch is false, while the second asserts that the branch is true. When the branch is executed, one of the two predicates will be true. The *branches* scheme applies to `if` statements, the branches governing `while` and `for` loops and the branches implied by the logical (`&&`, `||`) and conditional (`?`, `:`) operators [9, 10, 11].

Given a list of CBI predicates ranked by the *Importance* suspiciousness estimate (described in Section 3.2.4), statements are ranked according to the following:

1. For each statement, *stmt*, identify the corresponding predicate with the highest suspiciousness estimate, *stmt_{high}*.
2. Move the statement, *stmt* and highest suspiciousness estimate, *stmt_{high}*, to set *ST*.
3. Rank the statements in *ST* in descending order by suspiciousness estimate.

Exploratory Software Predictor (ESP)

The predicates employed in ESP are a superset of those employed in CBI. Along with the static *single variable*, static *scalar pairs* and *branches* instrumentation schemes employed in CBI, ESP also employs the elastic *single variable* and the elastic *scalar pairs* instrumentation schemes described in sections 3.3.1 and 3.3.2. The suspiciousness of all of these predicates and the ranking of the program statements in ESP are computed in the same manner as in CBI.

Tarantula

In contrast to the predicative-level statistical debuggers CBI and ESP, Tarantula is a statement-level statistical debugger. For a given statement, *stmt*, and a subject program test suite, Tarantula profiles the following:

- f_{stmt} - the number of failing test cases that cover (execute) statement *stmt*
- s_{stmt} - the number of successful test cases that cover (execute) statement *stmt*
- f - number of failing test cases
- s - number of successful test cases

The suspiciousness of the statement, *stmt*, is estimated in Tarantula by aggregating these terms using Equation 3.9. Once the suspiciousness of each statement that is executed in the test suite is estimated, all of the executed statements are ranked in descending order of suspiciousness [7].

$$Tarantula = \frac{\frac{f_{stmt}}{f}}{\frac{f_{stmt}}{f} + \frac{s_{stmt}}{s}} \quad (3.9)$$

Interesting Value Map Pairs (IVMP)

The IVMP debugger takes a different approach to fault localization than ESP, CBI and Tarantula. Instead of passively profiling a subject program to estimate predicate or statement suspiciousness, IVMP alters variable values in failing test cases. The goal is to find alternative values that cause failing test cases to become successful (passing) test cases.

IVMP estimates the suspiciousness of a statement, *stmt*, by tracking the number of failing test cases in which the alteration of the value of a variable within *stmt* creates a successful (passing) test case. Once the suspiciousness of each statement is estimated, the statements are ranked in descending order of suspiciousness. Any ties among statements with the same suspiciousness estimate are broken using the Tarantula suspiciousness estimate shown in Equation 3.9.

Table 3.5: Number (percentage) of faulty version ranked statement lists in each score range for all approaches for all subject programs.

Cost	Tarantula # (%) of Programs	IVMP # (%) of Programs	ESP # (%) of Programs	CBI # (%) of Programs
0 - 1%	13 (3.96%)	50 (15.24%)	43 (13.11%)	37 (11.28%)
1 - 10%	81 (24.70%)	143 (43.60%)	138 (42.07%)	120 (36.59%)
10 - 20%	61 (18.60%)	44 (13.41%)	53 (16.16%)	46 (14.02%)
20 - 30%	28 (8.54%)	16 (4.88%)	24 (7.32%)	26 (7.93%)
30 - 40%	36 (10.98%)	19 (5.79%)	21 (6.40%)	19 (5.80%)
40 - 50%	33 (10.06 %)	10 (3.05%)	10 (3.05%)	26 (7.93%)
50 - 60%	8 (2.43%)	10 (3.05%)	10 (3.05%)	10 (3.05%)
60 - 70%	12 (3.66%)	13 (3.96%)	8 (2.44%)	11 (3.35%)
70 - 80%	13 (3.96%)	6 (1.83%)	9 (2.74%)	18 (5.49%)
80 - 90%	10 (3.05%)	9 (2.74%)	9 (2.74%)	12 (3.66%)
90 - 100%	33 (10.06%)	8 (2.44%)	3 (0.91%)	3 (0.91%)

3.4.3 Effectiveness Studies

To study the effectiveness of each fault localization approach, an established cost metric is employed [5, 20, 31, 32, 104]. Given the ranked set of statements, the metric *Cost* measures the percentage of executed statements a developer must examine before encountering the faulty statement. If there are ties, it is assumed that the developer must examine all of the tied statements. For example, if there are n executed statements in a program and all n statements have the same suspiciousness estimate, it is assumed that the developer must examine all n statements. Therefore the *Cost* of finding the fault is 100%. A lower score is preferable because it means that fewer statements must be considered before the faulty statement is found.

The effectiveness of the four fault localization approaches on the subject programs included in the evaluation is shown in Table 3.5. Each row in Table 3.5 shows a *Cost* range, and the number (and percentage) of subject programs for each approach that incur the specified *Cost*. Figure 3.2 provides a graphical view of this data. In Figure 3.2 the x-axis represents the lower bound of each *Cost* range and the y-axis represents the percentage of subject programs where a *Cost* less than or equal to the upper bound is incurred. This presentation of data follows the established convention in the fault localization community [7, 12, 22, 23].

ESP vs. CBI

ESP performs better than CBI for the subject programs included in the evaluation. ESP assigned a lower rank than CBI to statements containing faults in only nine of 328 faulty program versions. In these instances the elastic predicates with higher suspiciousness estimates do not localize the faulty statement as well as their static counterparts with lower suspiciousness estimates. These are outlying data points in the evaluation (<3% of all faulty versions). For the vast majority of the faulty versions, identifying predicates with high suspiciousness estimates leads to the effective localization of faults.

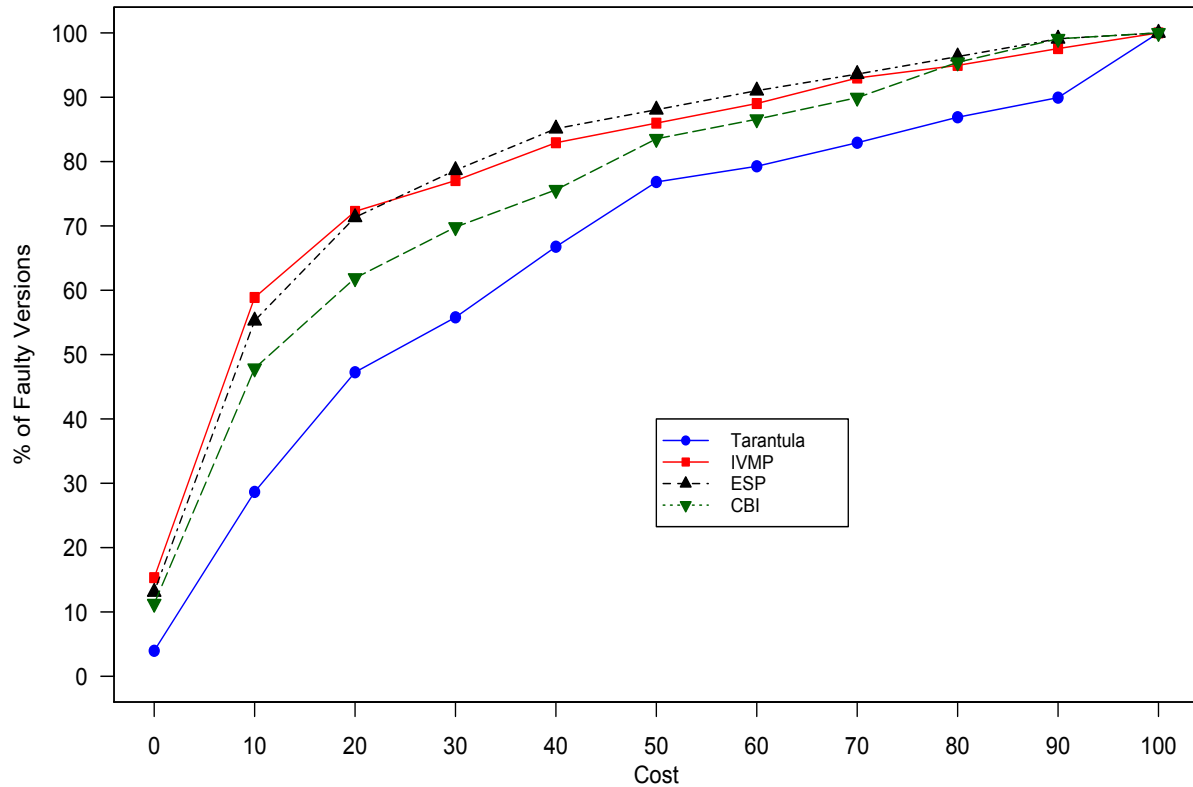


Figure 3.2: Comparison of the effectiveness of all the approaches. Higher and further left is better.

ESP vs. IVMP

IVMP performs better than ESP for the majority of the programs listed in Table 3.5. In the programs that did not perform extensive floating-point computations (`cal`, `col`, `look`, `tr`, `uniq`, `print-tokens`, `print-tokens2`, `replace`, `schedule`, `schedule2`, `tcas`, `sed`, `space`, `bc`, `flex`, `grep`) IVMP had a *Cost* of 10% or lower 123 times while ESP only had a *Cost* of 10% or lower 87 times. However, ESP performed well for those programs which make extensive use of floating point computations (`spline`, `totinfo`, `gzip`, `bc`, `bates`, `heston`, `mc euro`, `g/g/1`, `m/m/c`, `mmpp/d/1`) and IVMP did not. For these programs it is very difficult for IVMP to perform state alterations that cause a failing test case to become successful [22, 23]. This difficulty is due to the level of precision in the floating-point computations that generate the program’s output. As a result the approach echoes the ranking system in Tarantula for most statements. In the evaluation, IVMP performs the same as or worse than Tarantula for 40 of the 54 faulty versions of the programs that make extensive use of

floating-point operations [22, 23].

ESP vs. Tarantula

Compared to ESP, Tarantula is not effective for fault localization. ESP is able to uniquely identify the statement containing the fault (assign it rank one) in 41 cases. Tarantula is only able to do so in seven cases. Even though the ESP approach was able to uniquely identify the faulty statement in 41 cases, only 36 of those cases yielded a *Cost* of 1% or less because in the `tcas` program the number of statements executed in failing test cases was too few to yield a score of 1% or less. Tarantula is the only technique in the evaluation that does not consider variable values within an executing program. Instead it only analyzes if a statement was executed. This limits the effectiveness of the technique but enables it to be the most efficient technique evaluated. The efficiency of Tarantula and the other fault localization approaches is evaluated further in Section 3.4.8.

3.4.4 Widely Used Simulations

Table 3.6 highlights how effectively ESP localizes faults, compared to the other approaches, for the widely used simulations included in the evaluation. For each simulation and approach, the rank of the statement containing the fault is shown in Table 3.6. For each simulation the best rank is shown in bold and italicized.

Each of the simulations contains one fault that reflects a documented error that has been observed by subject matter experts (SMEs). For example, the Bates stochastic volatility jump-diffusion pricing model (`bates`) must be calibrated to previous data before it is employed to make price predictions for the future. However, if the absolute price error is minimized during calibration instead of the relative price error, the model produces an error [105]. Similarly, the implementation of the Heston stochastic volatility model (`heston`) reflects documented issues in the computation of the logarithms for complex numbers [106]. The pricing model of European Barrier Options (`mc euro`) contains an error in computation of bank offering rates [107]. The `um-olsr` protocol used with the `ns2` network simulator contains a documented (and now patched) error in the degree method [108, 109]. In the 2.19b version of the `ns2` network simulator, which can be used to model bandwidth for implementations of the TCP protocol, there is a fault, which incorrectly tracks the number of nodes in the network [110, 111].

The three queueing simulations included in the evaluation each contain published faults related to the misuse of uncertainty [112] (each of these simulations is described further in [113]). The first is a `g/g/1` queueing simulation employing a normal distribution (with infinite tails) when a hump-shaped distribution with values strictly greater than zero is intended. The second is a `m/m/c` queueing simulation with a misused

Table 3.6: Rank of faulty statement for all approaches for the widely used simulations.

Name	ESP	CBI	IVMP	Tarantula
bates	3	56	42	78
heston	1	16	68	144
mc euro	5	38	35	45
um-olsr	23	196	87	268
ns2	21	107	145	241
g/g/1	1	1	296	314
m/m/c	12	68	157	157
mmpp/d/1	13	69	64	213

Poisson distribution. The third is a `mmpp/d/1` queueing simulation with an incorrectly bound loop due to the misuse of a random variable.

IVMP performs poorly for most of the included simulations. This is attributed to two factors: (1) floating-point output from most of the simulations and (2) the extensive use of stochastics within several of the simulations. The first factor was evident in the previous portion of the evaluation. Recall, it is very difficult for IVMP to perform state alterations that cause a failing test case to pass in subject programs such as `totinfo` due to the level of precision in the floating-point computations that generate the program's output. This causes IVMP to echo Tarantula's ranking for most statements [22, 23]. The vast majority of these simulations include some floating-point output which creates the same issue for IVMP.

The second factor for IVMP's poor effectiveness is evident for the simulations that make the most use of stochastic distributions `mc euro`, `g/g/1` and `m/m/c`. In these simulations it is very possible that a test case will pass one time it is executed and fail another time. This creates trouble for a state-altering fault localization approach such as IVMP. IVMP relies on variable values in a failing test case to be replayed and altered to attempt to create a successful (passing) test case. However, the extensive use of stochastic distributions in `mc euro`, `g/g/1` and `m/m/c` causes some variable values to vary from execution to execution. Furthermore, when IVMP attempts to replay and alter a failing execution it is possible for the execution to become successful (pass) without alteration.

From this evaluation we hypothesize that stochastics significantly affect IVMP's analysis capabilities. To test this hypothesis the seed for each of the stochastic distributions used in the three simulations `mc euro`, `g/g/1` and `m/m/c` is fixed. Given a fixed seed, IVMP becomes approximately twice as effective as when the seed is not fixed for the three simulations. However, fixing a seed for each stochastic distribution still does not make IVMP as effective as ESP, and it limits the utility of the simulation for any user who has likely included the distribution to reflect model uncertainties.

3.4.5 Sparse Sampling

Recall from Section 3.3.3 that ESP and CBI use source code instrumentation to profile subject program predicates. The instrumentation, profiling and subsequent analysis adds overhead to subject program execution. In CBI this overhead can be limited by employing sparse random sampling of the instrumented predicates rather than always profiling every predicate. The sampling is unbiased, collecting a representative subset of all of the predicates across the subject program test suite. To ensure sufficient data collection, CBI relies upon the large user communities of the software in which it is deployed. The result is an effective approach to isolating faults in general-purpose software with wide distribution [9, 10, 11].

ESP is not designed to meet the same goals. In the most common use case, ESP is deployed as a stand-alone fault localization tool for a single user or SME. In this use case, the goal is to identify failure-predicting predicates as effectively as possible for the test cases provided. As a result, sparse random sampling is not used in ESP. However, in order to explore the environments in which ESP is useful it is important to evaluate elastic predicates in the context of sparse random sampling. The goal is to determine the extent to which the uncertainty introduced by sampling reduces the effectiveness of elastic predicates and thus ESP.

The *Costs* of one hundred executions of the subject programs included in the evaluation under ESP (striped) and CBI (non-striped) with sampling rates from 1/10 to 1/10,000 are plotted in Figure 3.3. The bottom and top of each box in Figure 3.3 represent the lower and upper quartile *Cost* and the black band is the median *Cost*. The whiskers extend to the lowest and highest *Cost*.

Figure 3.3 shows that the effectiveness of CBI remains more stable under sampling rates of 1/10 and 1/100 than the effectiveness of ESP. Once sampling is introduced the median *Cost* of localizing a fault in ESP increases significantly. While ESP continues to remain more effective than CBI by an absolute margin, the relative difference in effectiveness between the two approaches narrows. At a sampling rate of 1/1,000 both ESP and CBI begin to become significantly less effective.

The performance of ESP at a sampling rate of 1/10,000 reveals a trend: sufficiently infrequent sampling rates will reduce the effectiveness of ESP to that of CBI. In these cases the mean and standard deviation of each program point is based on so little data that the resulting elastic predicates are no better, and often worse, than the static and uniform predicates at predicting program failure. However, ESPs performance under more frequent sampling rates shows that elastic predicates do not require an exact calculation of the mean and standard deviation of values at each program point. Even at infrequent rates like 1/1,000, estimations of the mean and standard deviation result in effective failure-predicting predicates.

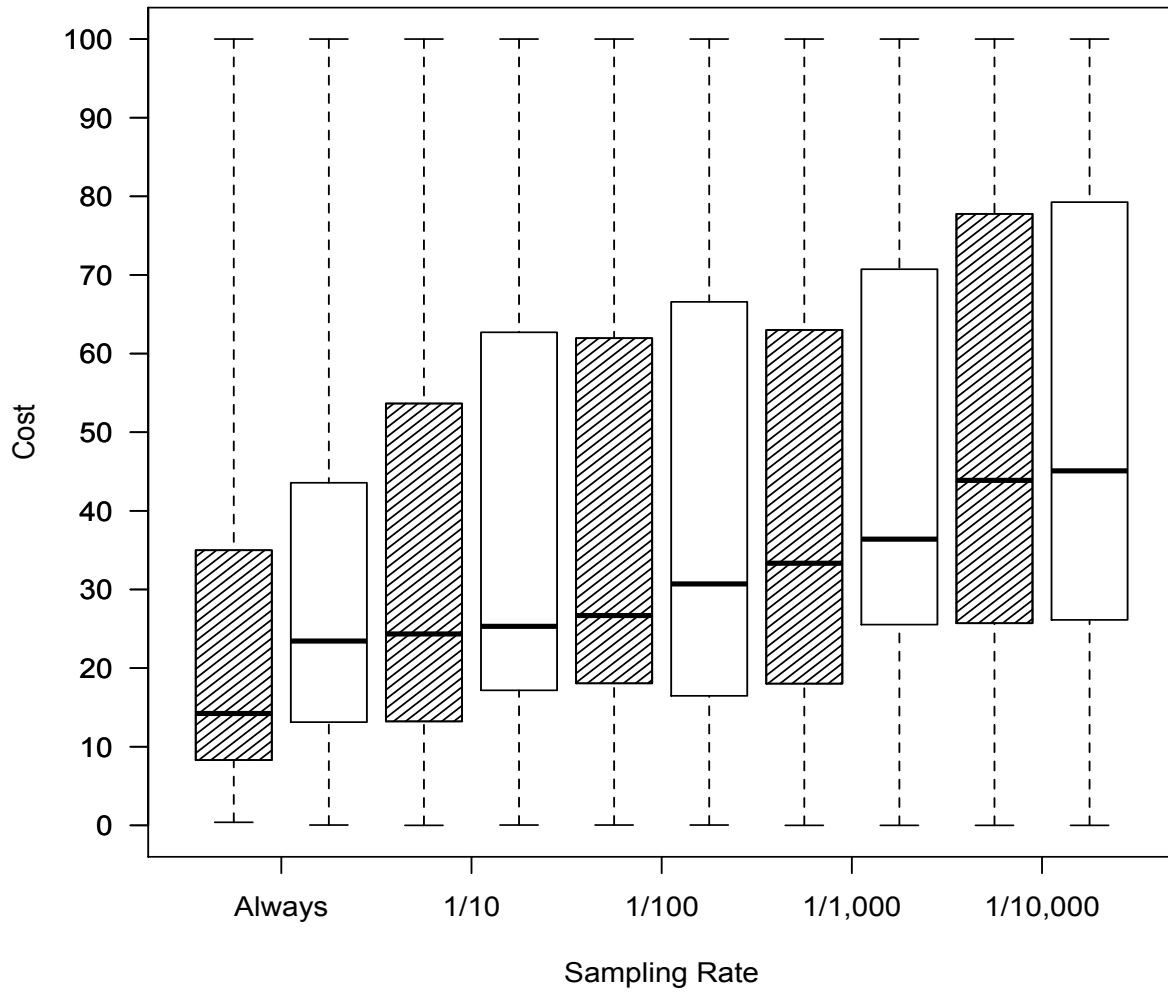


Figure 3.3: CBI (non-striped) and ESP (striped) under sparse sampling rates for the subject programs.

3.4.6 Incomplete Test Suites

Uncertainty in data collection can also be introduced through an incomplete or sparse test suite. Figure 3.4 summarizes the effect of incomplete test suites on the elastic predicates in ESP (striped) compared to the static predicates in CBI (non-striped). One hundred incomplete test suites from the original test suite for each subject program were chosen at random, forming test suites at $1/5^{\text{th}}$, $2/5^{\text{th}}$, $3/5^{\text{th}}$ and $4/5^{\text{th}}$ the size of the original suite. Each test case was chosen with uniform random probability without replacement and if the resulting sparse test suite did not contain at least ten failing test cases and at least twenty successful

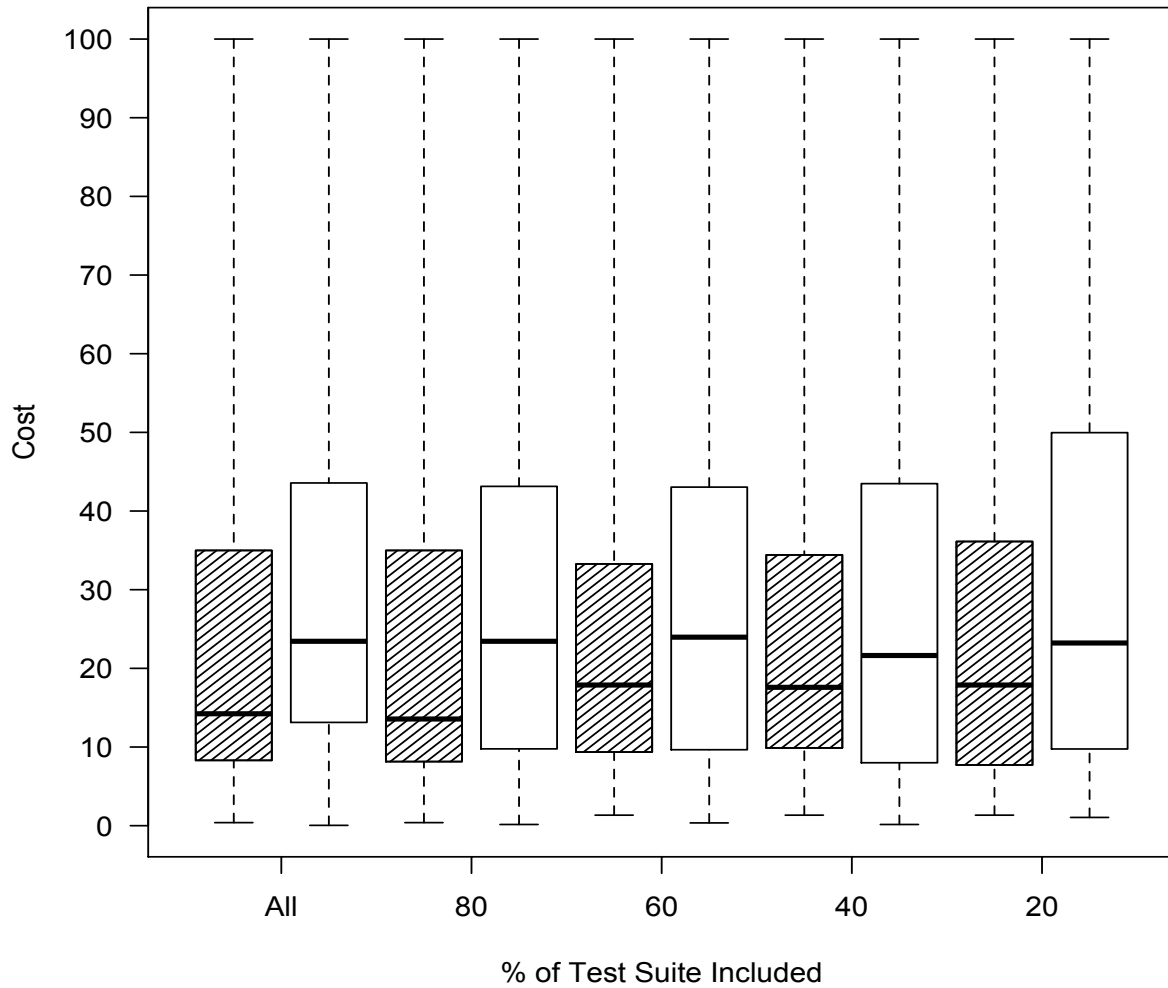


Figure 3.4: CBI (non-striped) and ESP (striped) with incomplete test suites for the subject programs.

(passing) test cases it was dissolved and the sparse test suite was reformed. Abreu et al. have shown that test suites formed with at least these numbers of successful (passing) and failing test cases do not have a strong effect on the effectiveness of existing fault localization techniques in comparison to larger test suites [12].

The effectiveness of ESP and CBI is stable across sparse test suites of different sizes for the subject programs included in the evaluation. Under the sparsest test suites included in the evaluation, $1/5^{\text{th}}$ of the complete test suite, both ESP and CBI show larger variation in effectiveness compared to the other test suite sizes. However, the median effectiveness of each technique at this test suite size is similar to the median of

each technique when all test cases are included. Overall, Figure 3.4 reveals that for the subject programs included in the evaluation, if the test suite formed features at least twenty successful (passing) test cases and ten failing test cases then the overall size of the test suite has little effect on the *Cost* incurred when applying ESP. In this regard ESP is similar to the statistical debuggers explored by Abreu et al. [12].

3.4.7 Suspiciousness Maximized Elastic Predicates

The suspiciousness of each elastic predicates used in ESP is not maximized. *Suspiciousness maximized elastic predicates* require numerical optimization to maximize the suspiciousness estimate for a specified program point, under a given instrumentation scheme. Unfortunately, forming *maximized elastic predicates* requires sorting and storing all of the values assigned to each instrumented program point. The overwhelming space and time required to perform the sorting and storing makes it infeasible to deploy *maximized elastic predicates* in practice. However, they are deployed in the statistical debugger, MAX, in this portion of our evaluation to determine the upper bound of effectiveness that is possible for *maximized elastic predicates* on a subset of the subject programs.

The subject programs included in this portion of the evaluation are: `cal`, `col`, `comm`, `look`, `spline`, `tr`, `uniq`, `print-tokens`, `print-tokens2`, `replace`, `schedule`, `schedule2`, `tcas` and `totinfo`. Each of these programs is small (512 lines of source code or fewer), making it possible to compute *maximized elastic predicates* for them. While this is a subset of the subject programs deployed in the full evaluation, it still reflects 14 different subject programs with 232 faulty versions.

The predicates employed in MAX are similar to those employed in ESP. MAX employs static *single variable* and *scalar pairs* predicates and branches predicates. However, MAX does not employ the elastic *single variable scalar pairs* predicates presented in sections 3.3.1 and 3.3.2. Instead for each program point, MAX instruments one *maximized elastic single variable predicate* and one *maximized elastic scalar pairs predicate*. Since the efficiency of MAX is not being considered, a brute force search strategy is employed to identify the one *single variable* predicate and the one *scalar pairs* predicate which maximizes the suspiciousness of the program point. These predicates reflect the conditions constructed from continuous invariants which are most correlated with failure in the subject program for the *Importance* suspiciousness estimate.

Table 3.7 and Figure 3.5 show the effectiveness of MAX compared to ESP and CBI for this portion of the evaluation. Once again, each row in Table 3.7 shows a *Cost* range, and the number (and percentage) of subject programs for each approach that incur the specified *Cost*. Figure 3.5 provides a graphical view of this data.

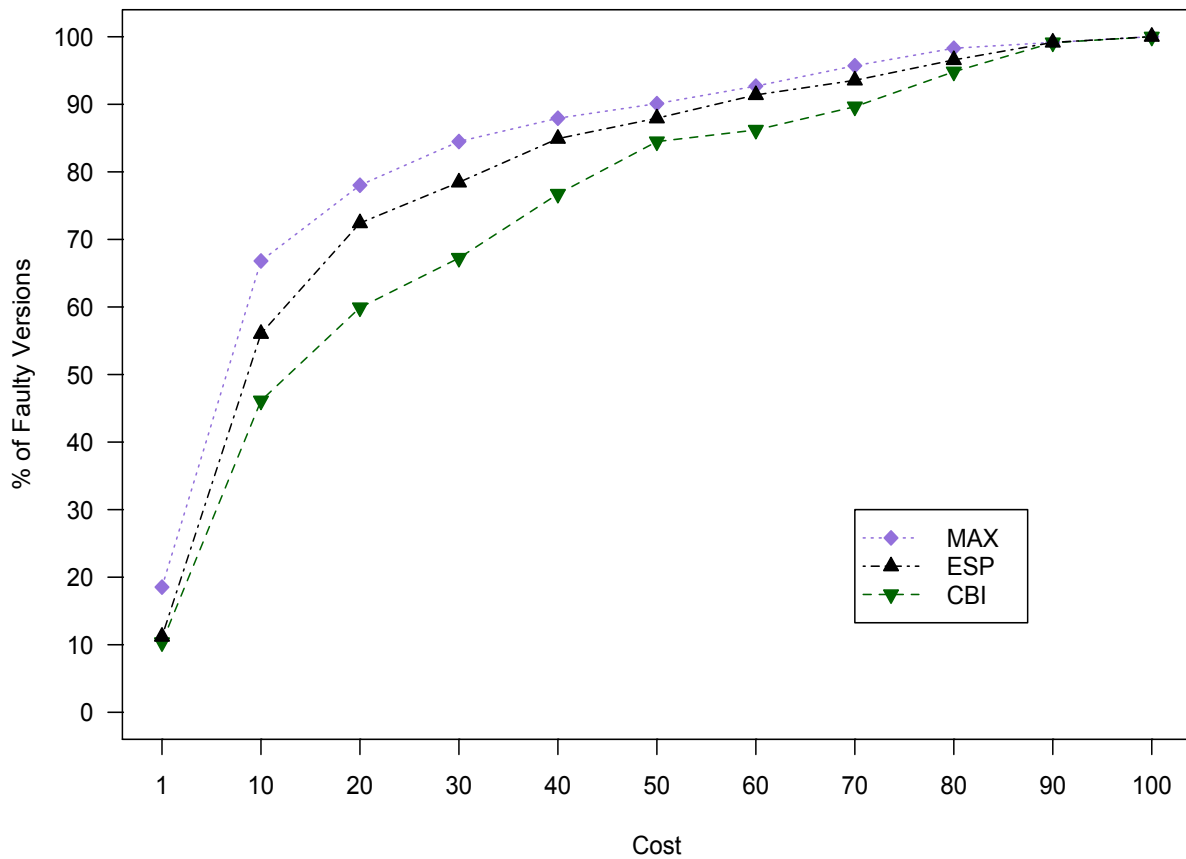


Figure 3.5: Comparison of the effectiveness of CBI, ESP and MAX. Higher and further left is better.

MAX and ESP perform similarly for the programs used in this portion of the evaluation which do not make extensive use of floating-point computations: (`cal`, `col`, `look`, `tr`, `uniq`, `print-tokens`, `print-tokens2`, `replace`, `schedule`, `schedule2`, `tcas`). In these programs MAX incurred a *Cost* of 1% or lower 27 times while ESP had a score of 1% or lower 21 times. However, for `totinfo` and `spline`, MAX incurred a *Cost* of 1% or lower 16 times while ESP only did so 5 times. This is not as discouraging as it appears. In 9 of the 11 cases where MAX incurred a *Cost* score of 1% or lower and ESP did not, ESP incurred a *Cost* of <10%.

Overall ESPs performance against MAX is encouraging. In 198 of the 232 faulty program versions, ESP approached the maximum effectiveness that is possible for a predicate-based statistical debugger. Furthermore, while ESP did not achieve the effectiveness of MAX in this portion of the evaluation it offered significant improvements over CBI despite none of the subject programs being simulations.

Table 3.7: Number (percentage) of faulty version ranked statement lists in each score range for CBI, ESP and MAX.

Cost	CBI # (%) of Programs	ESP # (%) of Programs	MAX # (%) of Programs
0 - 1%	24 (10.34%)	26 (11.21%)	43 (18.53%)
1 - 10%	83 (35.78%)	104 (44.83%)	112 (48.28%)
10 - 20%	32 (13.79%)	38 (16.38%)	26 (11.21%)
20 - 30%	17 (7.33%)	14 (6.03%)	15 (6.47%)
30 - 40%	22 (9.48%)	15 (6.47%)	8 (3.45%)
40 - 50%	18 (7.76%)	7 (3.02%)	5 (2.16%)
50 - 60%	4 (1.72%)	8 (3.45%)	6 (2.59%)
60 - 70%	8 (3.45%)	5 (2.16%)	7 (3.02%)
70 - 80%	12 (5.17%)	7 (3.02%)	6 (2.59%)
80 - 90%	10 (4.31%)	6 (2.59%)	2 (0.86%)
90 - 100%	2 (0.86%)	2 (0.86%)	2 (0.86%)

3.4.8 Efficiency

Here, the runtime efficiency of each of the fault localization approaches included in the evaluation is analyzed. Table 3.8 shows the wallclock time for ranking the statements for each faulty version of a subject program using Tarantula, IVMP and ESP and CBI. The wallclock time for employing *single variable* and *scalar pairs* predicates in ESP and CBI is shown separately. The *branches* instrumentation scheme is not included in this portion of the evaluation because it requires the same amount of time in ESP and CBI.

Figure 3.6 shows the data in Table 3.8 graphically. However, instead of measuring efficiency in terms of wallclock time, Figure 3.6 measures efficiency in terms of *Slowdown*. *Slowdown* is an efficiency measure which compares the wallclock time of an approach for a subject program relative to Tarantula’s wallclock time for the same subject program. The formula for *Slowdown* is shown in Equation 3.10. Because Tarantula is the most efficient approach in the evaluation the *Slowdown* of the other approaches is always >1 .

$$Slowdown = \frac{Runtime_{Other}}{Runtime_{Tarantula}} \quad (3.10)$$

Figure 3.6 reveals several trends in the relative efficiency of the *single variable* and *scalar pairs* predicates employed in ESP and CBI. While the combination of elastic and static *single variable* predicates in ESP does not result in constant *Slowdown*, the *Slowdown* is never greater than thirteen. Furthermore, when compared to static *single variable* predicates in CBI, the *Slowdown* of the *single variable* predicates in ESP is approximately constant (independent of the subject program to which it is applied) and only about a factor of five.

The *scalar pairs* predicates in CBI and ESP are significantly less efficient than their *single variable* counterparts. Employing static *scalar pairs* predicates in CBI can result in an execution time that is more

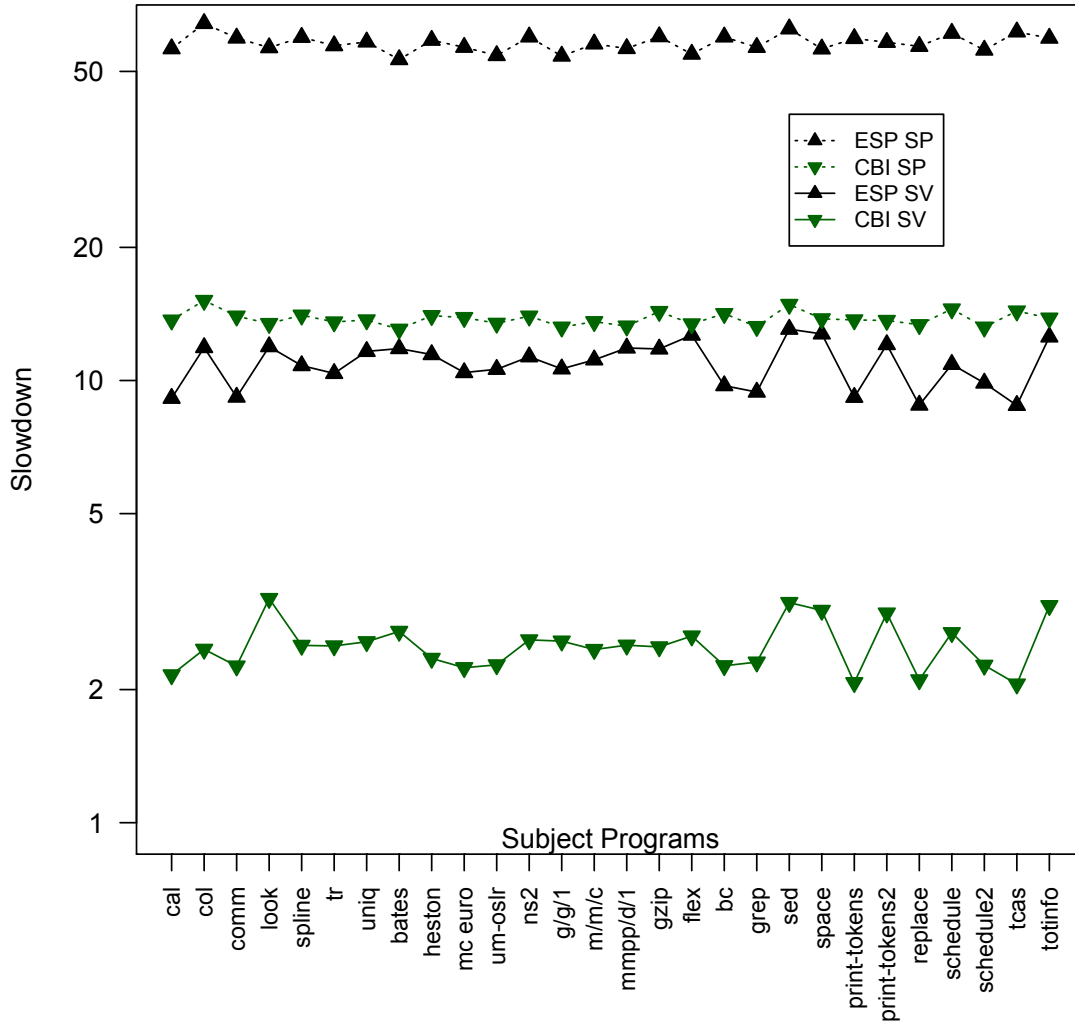


Figure 3.6: *Slowdown* relative to Tarantula for each competing approach. Note the log scale of the y-axis.

than fifteen times slower than applying Tarantula to the same subject program. Furthermore, for some of the subject programs included in the evaluation, employing both static and elastic *scalar pairs* predicates in ESP results in an execution time that is more than fifty times greater than that of Tarantula.

The *scalar pairs* predicates in ESP are not as inefficient as they first appear. When compared to *scalar pairs* predicates in CBI their *Slowdown* remains approximately constant and about five. This efficiency result ($\sim 5\times$ times slower than CBI) seems reasonable considering: (1) the significant improvement in the rank of faulty statements and (2) the ability to provide users with better failure predicting predicates.

The *Slowdown* for IVMP is not included in Figure 3.6 because IVMP is significantly less efficient than the

Table 3.8: Wallclock time (in seconds) required by each approach to execute all of the faulty versions of the specified subject program.

Name	Tarantula	CBI (SV)	ESP (SV)	CBI (SP)	ESP (SP)	IVMP
cal	9	20	85	127	525	2,240
col	8	20	97	124	527	1,543
comm	12	29	117	179	764	5,352
look	11	38	142	160	675	4,322
spline	23	59	253	329	1,400	32,110
tr	21	53	220	288	1,214	6,182
uniq	10	26	119	141	599	2,911
print-tokens	68	142	631	945	4,093	19,502
print-tokens2	50	150	607	688	2,930	14,210
replace	75	159	667	1,014	4,324	11,592
schedule	30	80	327	436	1,834	7,041
scheule2	30	68	296	396	1,677	5,278
tcas	12	25	108	177	759	978
totinfo	8	26	106	117	466	104,060
sed	738	2,325	9,651	10,980	46,114	242,410
space	244	737	3,110	3,364	13,718	102,536
bc	38	86	371	540	2,281	304,025
gzip	684	1,710	8,067	9,802	41,024	154,192
flex	1,064	2,813	13,480	14,291	58,231.27	134,924
grep	730	1,685	6,874	9,667	41,387	34,738
bates	35	95	417	461	1,878	23,234
heston	56	132	643	787	3,305	96,949
mc euro	40	90	421	561	2,295	104,358
um-olsr	30	69	325	413	1,668	14,911
ns2	80	209	915	1,131	4,842	24,598
g/g/1	23	60	249	309	1,270	22,559
m/m/c	18	46	210	256	1,091	40,345
mmp/d/1	29	75	353	395	1,680	57,593

other approaches. For the subject programs which make extensive use of floating-point computations (`spline`, `totinfo`, `gzip`, `bc`, `bates`, `heston`, `mc euro`, `g/g/1`, `m/m/c`, `mmp/d/1`) the *Slowdown* of IVMP approaches infinity. This drastic degradation is due to failing test case executions that IVMP repeatedly re-executes in an attempt to find state alterations resulting in a successful execution. For programs such as these, where it is difficult to perform state alterations on failing test cases to create successful test cases, IVMP is extremely inefficient [22].

3.4.9 Reduced Elastic Predicates

While ESP's efficiency compared to CBI seems reasonable given its improved fault localization capabilities, it is possible to improve the speed of ESP while maintaining most of its effectiveness. FAST ESP, the reduced predicate approach presented in this section, provides users with a more efficient version of ESP with only a slight reduction in effectiveness. FAST ESP offers *speedup* compared to ESP by only instrumenting a subset

Table 3.9: Groups of elastic predicates in ESP with similar suspiciousness estimates.

Group	Single Variable Elastic Predicates	Scalar Pairs Elastic Predicates
1	$x_y > (\mu_{x_y} + 3\sigma_{x_y})$ $(\mu_{x_y} + 3\sigma_{x_y}) \geq x_y > (\mu_{x_y} + 2\sigma_{x_y})$ $(\mu_{x_y} + 2\sigma_{x_y}) \geq x_y > (\mu_{x_y} + \sigma_{x_y})$	$x_y - q_i > (\mu_{x_y - q_i} + 3\sigma_{x_y - q_i})$ $(\mu_{x_y - q_i} + 3\sigma_{x_y - q_i}) \geq x_y - q_i > (\mu_{x_y - q_i} + 2\sigma_{x_y - q_i})$ $(\mu_{x_y - q_i} + 2\sigma_{x_y - q_i}) \geq x_y - q_i > (\mu_{x_y - q_i} + \sigma_{x_y - q_i})$
2	$(\mu_{x_y} + \sigma_{x_y}) \geq x_y > \mu_{x_y}$ $(\mu_{x_y} - \sigma_{x_y}) \leq x_y < \mu_{x_y}$	$(\mu_{x_y - q_i} + \sigma_{x_y - q_i}) \geq x_y - q_i > \mu_{x_y - q_i}$ $(\mu_{x_y - q_i} - \sigma_{x_y - q_i}) \leq x_y - q_i < \mu_{x_y - q_i}$
3	$(\mu_{x_y} - 2\sigma_{x_y}) \leq x_y < (\mu_{x_y} - \sigma_{x_y})$ $(\mu_{x_y} - 3\sigma_{x_y}) \leq x_y < (\mu_{x_y} - 2\sigma_{x_y})$ $x_y < (\mu_{x_y} - 3\sigma_{x_y})$	$(\mu_{x_y - q_i} - 2\sigma_{x_y - q_i}) \leq x_y - q_i < (\mu_{x_y - q_i} - \sigma_{x_y - q_i})$ $(\mu_{x_y - q_i} - 3\sigma_{x_y - q_i}) \leq x_y - q_i < (\mu_{x_y - q_i} - 2\sigma_{x_y - q_i})$ $x_y - q_i < (\mu_{x_y - q_i} - 3\sigma_{x_y - q_i})$
4	$\mu_{x_y} = x_y$	$\mu_{x_y - q_i} = x_y - q_i$

Table 3.10: FAST ESP elastic predicates.

Single Variable Elastic Predicates	Scalar Pairs Elastic Predicates
$x_y > (\mu_{x_y} + \sigma_{x_y})$ $(\mu_{x_y} + \sigma_{x_y}) \geq x_y \geq (\mu_{x_y} - \sigma_{x_y})$ $x_y < (\mu_{x_y} - \sigma_{x_y})$	$x_y - q_i > (\mu_{x_y - q_i} + \sigma_{x_y - q_i})$ $(\mu_{x_y - q_i} + \sigma_{x_y - q_i}) \geq x_y - q_i \geq (\mu_{x_y - q_i} - \sigma_{x_y - q_i})$ $x_y - q_i < (\mu_{x_y - q_i} - \sigma_{x_y - q_i})$

of the elastic *single variable* and *scalar pairs* predicates that ESP does.

Recall, ESP employs the static and elastic *single variable* predicates described in 3.3.1, the static and elastic *scalar pairs* predicates described in 3.3.2 and the *branches* predicates described in Section 3.4.2. The elastic predicates in ESP partition the value of instrumented program points for three standard deviations above and below the mean of μ_{x_y} and $\mu_{x_y - q_i}$ respectively, where x is a variable, q is a same-typed variable in x 's scope, and y and i are the statements corresponding to x and q respectively. The result of this partitioning is nine *single variable* elastic predicates and nine *scalar pairs* elastic predicates for each program point.

However, these eighteen elastic predicates are not always necessary to localize the faults in the subject programs included in the evaluation. Often, for a given program point, the suspiciousness estimates for several of the elastic *single variable* predicates are correlated. Similarly, the suspiciousness estimates for several of the *scalar pairs* elastic predicates for a program point are correlated. Table 3.9 shows the groups of *single variable* and *scalar pairs* elastic predicates which are the most correlated for a given program point.

FAST ESP, exploits the groups of highly correlated predicates shown in Table 3.9 by aggregating all of the correlated predicates in one group, for one instrumentation scheme, into one predicate. The resulting set of *single variable* and *scalar pairs* elastic predicates employed in FAST ESP is shown in Table 3.10. It is important to note that *single variable* elastic predicate $\mu_{x_y} = x_y$ and the scalar pairs elastic predicate $\mu_{x_y - q_i} = x_y - q_i$ are not correlated with any other predicates because they are the only predicates in their respective instrumentation schemes to employ the equality operator ($=$). Within FAST ESP these equality predicates are aggregated into Table 3.9's Group 2 with the other elastic predicates from the same instrumentation scheme.

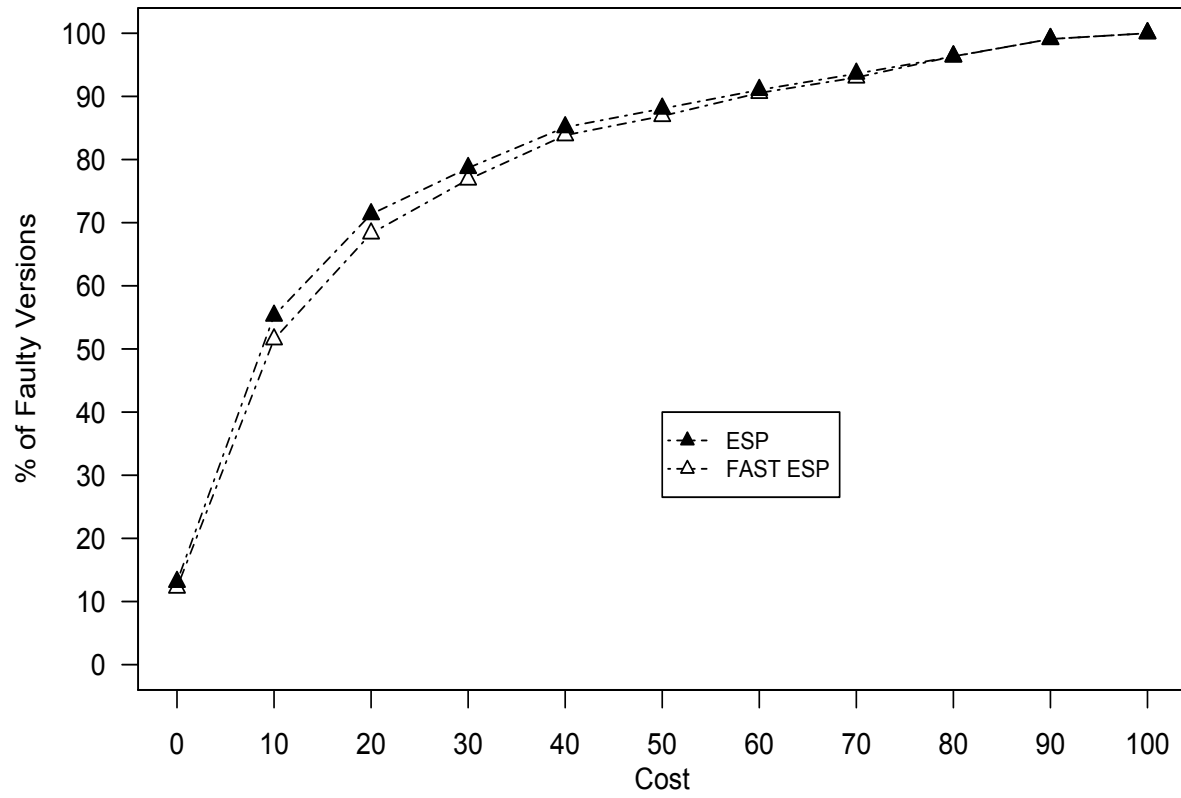


Figure 3.7: Comparison of the effectiveness of ESP and FAST ESP. Higher and further left is better.

Table 3.11: Number (percentage) of faulty version ranked statement lists in each score range for FAST ESP and ESP.

Cost	FAST ESP # (%) of Programs	ESP # (%) of Programs
0 - 1%	40 (12.20%)	43 (13.11%)
1 - 10%	129 (39.32%)	138 (42.07%)
10 - 20%	55 (16.77%)	53 (16.16%)
20 - 30%	28 (8.54%)	24 (7.32%)
30 - 40%	23 (7.01%)	21 (6.40%)
40 - 50%	10 (3.05%)	10 (3.05%)
50 - 60%	12 (3.66%)	10 (3.05%)
60 - 70%	8 (2.44%)	8 (2.44%)
70 - 80%	11 (3.35%)	9 (2.74%)
80 - 90%	9 (2.74%)	9 (2.74%)
90 - 100%	3 (0.91%)	3 (0.91%)

Table 3.12: Wallclock time (in seconds) required by ESP and FAST ESP to execute all of the faulty versions of the specified subject program.

Name	FAST ESP (SV)	ESP (SV)	FAST ESP (SP)	ESP (SP)
cal	69	85	443	525
col	91	97	464	527
comm	105	117	671	764
look	136	142	554	675
spline	226	253	1,241	1,400
tr	214	220	1,131	1,214
uniq	115	119	502	599
print-tokens	604	631	3,897	4,093
print-tokens2	493	607	2,558	2,930
replace	555	667	3,881	4,324
schedule	266	327	1,597	1,834
scheule2	248	296	1,576	1,677
tcas	102	108	708	759
totinfo	100	106	414	466
sed	9,098	9,651	42,048	46,114
space	2,739	3,110	11,788	13,718
bc	313	371	1,925	2,281
gzip	7,449	8,067	37,920	41,024
flex	11,001	13,480	57,259	58,231
grep	5,766	6,874	1,920	41,387
bates	385	417	1,819	1,878
heston	619	643	3,033	3,305
mc euro	381	421	2,004	2,295
um-olsr	320	325	1,394	1,668
ns2	859	915	4,160	4,842
g/g/1	205	249	1,253	1,270
m/m/c	175	210	950	1,091
mmpp/d/1	353	353	1,660	1,680

Table 3.11 and Figure 3.7 show the effectiveness of FAST ESP compared to ESP for this portion of the evaluation. Once again, each row in Table 3.11 shows a *Cost* range, and the number (and percentage) of subject programs for each approach that incur the specified *Cost*. Figure 3.5 provides a graphical view of this data.

The effectiveness of FAST ESP is very similar to that of ESP. The major difference between the two debuggers is that ESP is capable of identifying faults in 181 of the subject program versions while incurring 10% *Cost* or less, while FAST ESP identifies 169 faults within the same *Cost* range. Thus, for developers whose ultimate goal is effectiveness, ESP remains a better choice than FAST ESP. However, because FAST ESP employs 12 fewer predicates than ESP for each instrumented program point it is more efficient.

Figure 3.8 and Table 3.12 show the difference in efficiency between the *single variable* and *scalar pairs* elastic predicates employed in FAST ESP and ESP. Once again, the *Slowdown* incurred by each approach is computed relative to Tarantula’s runtime. This relative efficiency measure is shown in Equation 3.10.

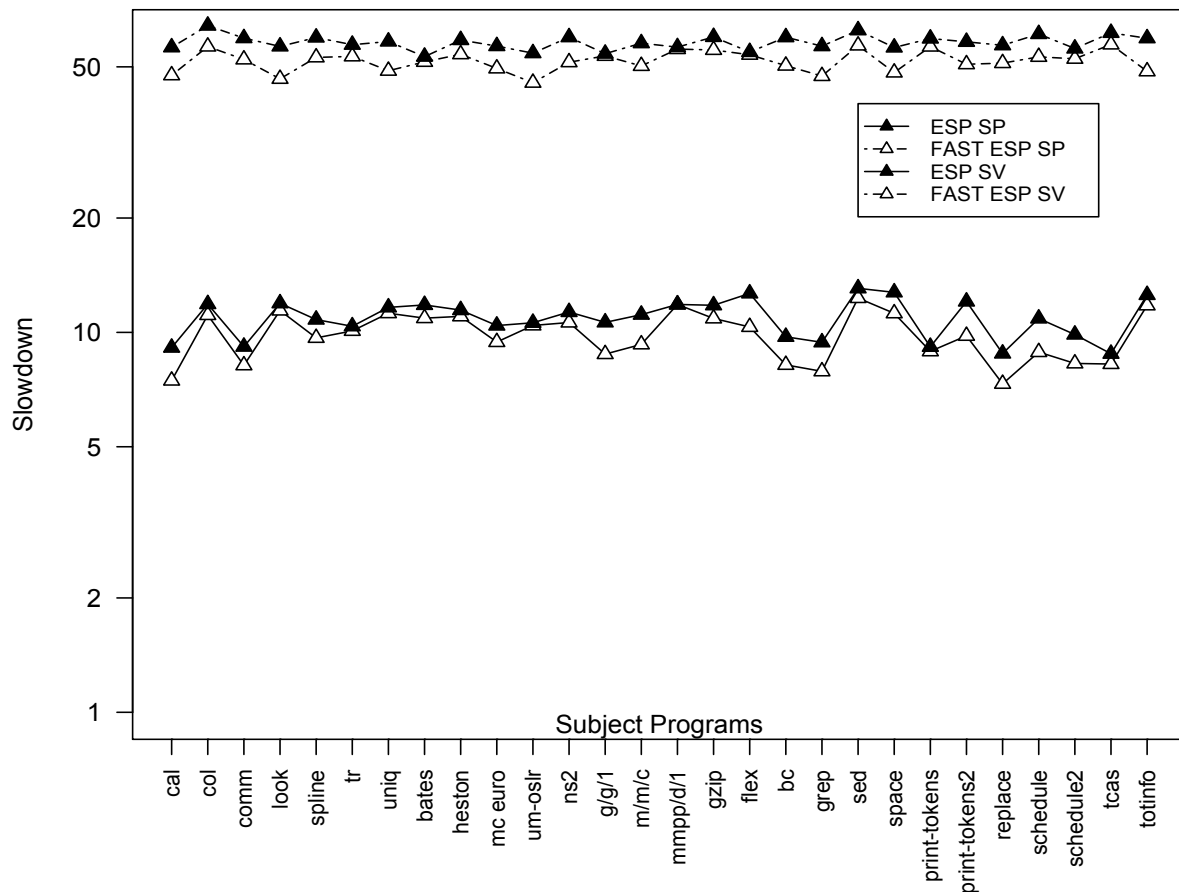


Figure 3.8: Comparison of the efficiency of ESP and FAST ESP. Note the log scale of the y-axis.

On average the *single variable* and *scalar pairs* instrumentation schemes in FAST ESP incur about 15% less *Slowdown* than the respective schemes in ESP. As a result FAST ESP is only slightly more than four (4.25) times slower than CBI, while ESP is about five times slower than CBI. This improvement is significant. Employing the FAST ESP elastic *scalar pairs* predicates on all of the faulty versions of the subject program `sed` requires fifteen hours of wallclock time, compared to ESP which requires seventeen total hours. Furthermore, there is no difference in effectiveness between the approaches for the faulty versions of `sed`.

Despite these improvements in efficiency, developers whose ultimate goal is effectiveness should employ the elastic predicates in ESP instead of the reduced set in FAST ESP. ESP is more effective than FAST ESP at localizing faults while incurring a *Cost* of 10% or less for the subject programs in the evaluation. Furthermore, the additional 15% *Slowdown* incurred by ESP only requires machine time not developer time.

If developers can remain productive while ESP is running, overall efficiency will be improved because the developer is given a more effective list of ranked predicates to identify faults.

3.4.10 Validity

Internal, external, and construct validity threats affect the evaluation of the elastic predicates included in ESP. Threats to internal validity arise when factors affect the dependent variables without the evaluators' knowledge. It is possible that some flaws in the implementation of CBI, IVMP and Tarantula could have affected the results of the evaluation. However, the results for the subject programs included in the evaluation for the competing fault localization approaches closely match published results from other researchers [7, 13, 22]. Threats to external validity occur when the results of the evaluation cannot be generalized. Although the evaluation was performed on 28 subject programs with a total of 328 versions, the effectiveness of ESP and elastic predicates cannot be generalized to other faults in other software subjects. However, the results for the 28 subject programs and 328 faulty versions demonstrate the power of elastic predicates in comparison to static predicates. Threats to construct validity concern the appropriateness of the *Cost* metric used in the evaluation. While the *Cost* metric is well established in the fault localization community, more human user studies into how useful developers find statement-ranking metrics need to be performed [114]. However, the more accurate fault-localization methods are, the more meaningful such human user studies will become.

3.4.11 Multiple Faults

Several versions of the subject programs in the evaluation contain multiple faults. This is not uncommon; faulty programs often feature multiple faults. In the evaluation, each fault localization approach is only required to identify one fault per program. This allowed ESP to be conservatively evaluated against the best available alternatives.

However it is important to note that statistical debugging approaches such as CBI and ESP remain effective in the face of subject programs with multiple faults, while state-altering approaches such as IVMP do not. It is difficult for state-altering approaches to differentiate among multiple faults in a program because they have trouble identifying alterations in failing test cases that have different effects on the program output [22]. Several modifications to state-altering fault localization approaches have been suggested to address this issue, however, these modifications can make the approaches even less efficient [23].

ESP and CBI use the following established algorithm to guarantee a developer at least one failure predicting predicate for each fault that is present in a faulty subject program [9, 10, 11]. The result is that

CBI and ESP are applicable to a larger set of subject programs and faults than state-altering approaches because they are more robust and effective.

1. Rank each predicate in descending order by suspiciousness estimate.
2. Remove the top-ranked predicate p and discard all test cases where the p was found to be true.
3. Repeat steps 1 and 2 until the set of test cases is empty or the set of predicates is empty.

3.5 Chapter Summary

Previous approaches to predicate-level statistical debugging assumed discrete data types and static predicates, which precluded practical consideration of programs such as simulations and computational models, which employ floating-point computations and continuous stochastic distributions. The elastic predicates presented in this chapter directly address this limitation. Elastic predicates employ the mean (μ) and standard deviation (σ) of variable values to create predicates that can adapt to values observed in test cases. The profiling of variable values makes elastic predicates fundamentally different from the static predicates used in existing tools, such as CBI. Existing static predicates are generated prior to test case execution and as a result unable to adapt to variable values.

Our statistical debugger ESP complements the existing static predicates in CBI with elastic predicates. ESP is more effective than any of the other statistical debugging approaches included in the evaluation and is shown to be as effective and more efficient than the state-altering approach IVMP. Furthermore, the evaluation shows that the elastic predicates in ESP remain effective in the face of sparse sampling rates and incomplete test suites.

While our elastic predicates do not employ numerical optimization to maximize the suspiciousness of each instrumented program point as *maximized elastic predicates* do, they achieve some of the improvements that are possible, without incurring related loss of efficiency. Overall, elastic predicates offer significant improvements to statistical debuggers for localizing faults in software which employ floating-point computations and continuous stochastic distributions. Furthermore, elastic predicates offer modest improvements to statistical debuggers for localizing faults in general purpose software.

Chapter 4

Reducing Confounding Bias

The effectiveness of predicate-level statistical debuggers can be improved through multiple avenues. The *elastic predicates* presented in Chapter 3 improve statistical debuggers by creating predicates that more effectively localize causes of failure. However, statistical debuggers can also be improved by estimating the suspiciousness of a predicate from collected data more effectively. In this chapter an approach to improve the effectiveness of each of the suspiciousness estimates from Section 3.2 for both general-purpose software and exploratory simulations is presented. The result is better fault localization capabilities for predicate-level statistical debuggers independent of the type of predicate (elastic or static) employed.

The suspiciousness estimates in Section 3.2 employ statistical techniques on observational data to determine the effect of individual predicates on program failure. However, all of these estimates are susceptible to confounding bias. *Confounding bias* occurs when an apparent causal effect of an event on an outcome may actually be due to an unknown confounding variable, which causes both the event and the outcome [92, 94]. The suspiciousness estimates in Section 3.2 are susceptible to confounding bias because they do not adequately *control for* the effects of other predicates on: (1) the suspiciousness of the predicate being estimated or (2) on the occurrence of program failure. An example helps elucidate how confounding bias occurs in these estimates.

4.1 Motivating Example

Consider the procedure `distance()` in Figure 4.1, which has a fault in Statement 5. The procedure should print the one dimensional Euclidean distance between two points x and y . However, for some test cases `distance()` prints out a negative number. The observational data collected for the truth of the static predicates for statements 5, 6, 7 and 8 is shown in Table 4.1.

```

1 int
2 distance(int x, int y)
3 {
4     int diff = x - y;
5     if (!(diff > 1)){ /* off by one */
6         int dist = 0;
7         dist = y - x;
8         return dist;
9     }
10    int dist = 0;
11    dist = x - y;
12    return dist;
13 }

```

Figure 4.1: Faulty function that calculates euclidean distance between two one-dimensional points .

The first two columns in Table 4.1 identify the statement number and condition tested in each of the static predicates. The third through the seventh columns identify the inputs for five test cases. The '1/0' entries within these columns indicate if the corresponding predicate was true in each test case. An entry of '1' means that the predicate was true and an entry of '0' means that the predicate was not true. The bottom row of Table 4.1 shows the Boolean outcome of executing each test case. A failing test case execution is denoted by 'FAIL' and a successful (or passing) test case execution is denoted by 'PASS'.

The two rightmost columns in Table 4.1 indicate the rank of each predicate based on two different suspiciousness estimates. The first suspiciousness estimate is the *specificity* measure described in Section 3.2.1. Recall, *specificity* is the sample ratio $f_p/(f_p + s_p)$ where, f_p is the number of tests for which p is true and the program fails and s_p is the number of tests for which p is true and the program succeeds. The second suspiciousness estimate, found in the rightmost column of table 4.1, is derived from a causal model which *controls for* the effects of other predicates on the suspiciousness of the predicate being estimated and on the occurrence of program failure. Recall, causal models are presented in detail in Section 2.7.

Table 4.1 shows that the *specificity* measure identifies three predicates which rank as the most suspicious predicates in the procedure `distance()` shown in Figure 4.1. These predicates are listed separately in Table 4.2. Recall, a predicate x specifying a condition corresponding to program statement y is abbreviated as x_y .

The first predicate in Table 4.2, $(\text{diff} > 0)_5$, reflects the location of the fault in the procedure `distance()`. However, the two other predicates in Table 4.2, which have the same suspiciousness rank as $(\text{diff} > 0)_5$ correspond to innocent statements. These two predicates have the same rank as the fault localizing predicate, despite their ties to innocent statements, because their truth is dependent on the condition $\text{diff} > 0$ in Statement 5. This dependence causes the predicates $(\text{dist} < 0)_7$ and $(\text{dist} < 0)_8$ to be true in every failing execution. Since the *specificity* estimate *does not control for* dependence among predicates, confounding bias exists and the two dependent predicates $(\text{dist} < 0)_7$ and $(\text{dist} < 0)_8$ receive the same rank as the fault

Table 4.1: Test cases and predicate data for procedure `distance()` in Figure 4.1.

Statement	Predicate	TC 1: {2,2}	TC 2: {5,4}	TC 3: {5,1}	TC 4: {-4,-2}	TC 5: {1,0}	<i>Specificity</i> Ranks	Causal Model Ranks
5	<code>diff = 0</code>	1	0	0	0	0	12	12
5	<code>diff > 0</code>	0	1	0	0	1	3	1
5	<code>diff < 0</code>	0	0	1	1	0	12	12
6	<code>dist = 0</code>	1	1	1	1	1	4	12
6	<code>dist > 0</code>	0	0	0	0	0	12	12
6	<code>dist < 0</code>	0	0	0	0	0	12	12
7	<code>dist = 0</code>	1	0	0	0	0	12	12
7	<code>dist > 0</code>	0	0	1	1	0	12	12
7	<code>dist < 0</code>	0	1	0	0	1	3	2
8	<code>dist = 0</code>	1	0	0	0	0	12	12
8	<code>dist > 0</code>	0	0	1	1	0	12	12
8	<code>dist < 0</code>	0	1	0	0	1	3	2
OUTCOME	——	PASS	FAIL	PASS	PASS	FAIL	——	——

Table 4.2: Most suspicious predicates by *specificity* estimate for the procedure `distance()` in Figure 4.1.

Statement	Predicate	Abbreviated Name
5	<code>diff > 0</code>	(<code>diff > 0</code>) ₅
7	<code>dist < 0</code>	(<code>dist < 0</code>) ₇
8	<code>dist < 0</code>	(<code>dist < 0</code>) ₈

localizing predicate (`diff > 0`)₅. This dependency related confounding bias is referred to as *control flow dependency* confounding bias.

However, another type of confounding bias exists in the *specificity* estimate used to rank the predicates in the procedure `distance()`. Additional confounding bias exists because the fault in Statement 5 is triggered before the predicates corresponding to statements 6, 7 and 8 are evaluated. This bias is evident in the suspiciousness rank (4) calculated from the *specificity* estimate for the predicate (`dist = 0`)₆, which corresponds to the innocent declaration of the variable `dist` in Statement 6.

While the condition `dist = 0` in Statement 6 is uncorrelated with failure, it is always true immediately following the execution of Statement 5 and thus is true in every failing test case. As a result the *specificity* estimate finds the predicate (`dist = 0`)₆ to be suspicious. However, it is important to note that the chance of a test case failing given that (`dist = 0`)₆ is true, is the same as the chance of a test case failing given that Statement 6 is executed. The inability to control for the difference between the chance of a test case failing given that a predicate is *true* and the chance of a test case failing given that the predicate is *evaluated* (*true or false*) results in a second type of confounding bias: *failure flow* confounding bias.

The factors that create *control flow dependency* confounding bias and *failure flow* confounding bias can be controlled for in a causal model. The causal model produces a suspiciousness estimate with reduced bias that more effectively ranks predicates for fault localization. In this example, the causal model predicate

rankings clearly identify one suspicious predicate that localizes the fault in the procedure `distance()` in Figure 4.1. This is shown in the right-hand column of Table 4.1. In Section 4.2 the factors that create *control flow dependency* confounding bias and *failure flow* confounding bias are identified and causal models which control for these factors to produce suspiciousness estimates with reduced bias are presented.

4.2 Controlling for Confounding Bias

Within the motivating example in Section 4.1, two different types of confounding bias are evident: *control flow dependency* confounding bias and *failure flow* confounding bias. While controlling for both of these confounding biases through a causal model to estimate suspiciousness for predicate-level statistical debuggers is our novel work, each of the biases has been previously explored, separately, in other research.

At the statement-level, Baah et al. have shown that in a subject program *control flow dependency* confounding bias is manifested on a given statement *stmt* through the execution of the statement immediately preceding *stmt* in the control flow graph [31, 32]. This statement is the *forward control flow predecessor* of *stmt*. The execution of the *forward control flow predecessor* causes statement-level suspiciousness estimates to overrate the influence of dependent statements, such as *stmt*, on the subject program’s failure. By employing a causal model to control for the *forward control flow predecessor*, Baah et al. improved the effectiveness of suspiciousness estimates for statement-level statistical debuggers [31, 32].

Inspired by Baah et al.’s statement-level work on reducing *control flow dependency* confounding bias, we define a method for reducing bias for the predicate-level. Our refocusing of this work poses challenges which require innovation. Our predicate-level adaptation, the challenges and the resulting innovation are described in Section 4.2.2.

While *control flow dependency* confounding bias has not been perviously explored at the predicate-level, *failure flow* confounding bias has. Liblit et al. have shown that *failure flow* confounding bias exists in the *specificity* suspiciousness estimate. Recall, the *specificity* measure estimates the probability of a subject program *Q* failing given that predicate *p* is true, $\Pr(Q \text{ fails} \mid p=\text{true})$, with the ratio: $f_p/(f_p + s_p)$ [10, 11]. This estimate of $\Pr(Q \text{ fails} \mid p=\text{true})$ is biased because once a fault in a program has been triggered, the program is much more likely to fail. Liblit et al. showed that under this circumstance subsequent predicates are more likely to be observed in failing executions and appear more suspicious than they actually are [10, 11]. Liblit et al.’s observation reflects *failure flow* confounding bias and it demonstrates that *failure flow* confounding bias must be identified at the predicate-level, not the statement-level.

To address *failure flow* confounding bias, Liblit et al. proposed the *Importance* measure. Recall from Section 3.2.4, the *Importance* measure estimates $\Pr(Q \text{ fails} \mid p=\text{true})$ with the difference $f_p/(f_p + s_p) -$

$f_{p\text{ obs}}/(f_{p\text{ obs}} + s_{p\text{ obs}})$. The terms $f_{p\text{ obs}}$ and $s_{p\text{ obs}}$ are the number of respective failing and succeeding test cases for which p is evaluated. Liblit et al.’s estimate of $\Pr(Q \text{ fails} \mid p=\text{true})$ measures not the chance that a predicate p implies failure, but how much difference it makes that the predicate p is observed to be true versus simply reaching the line where the predicate p is evaluated [10]. This correction is a heuristic to factor out the program flow once the fault is triggered within a subject program. It attempts to control for *failure flow* confounding bias. While Liblit et al.’s heuristic can be effective it does not employ a methodology, such as causal models, that is a proven solution for addressing confounding bias.

In Section 4.2.3, our causal model which reduces *control flow dependency* confounding bias and *failure flow* confounding bias is presented. The evaluation in Section 4.2.5 shows that employing this causal model to control for both confounding biases results in significantly more effective suspiciousness estimates for predicate-level statistical debuggers than any of the estimates described in Section 3.2, including Liblit et al.’s *Importance* estimate. Furthermore, the approach presented in Section 4.2.3 is more effective than our alternative approach proposed in Section 4.2.2, which only reduces *control flow dependency* confounding bias.

However, before any of our models for reducing confounding bias are presented, Section 4.2.1 reviews background information related to observational studies, confounding bias and causal models which is required to understand the approach to reducing bias taken in this chapter.

4.2.1 Observational Studies, Causal Models and Confounding Bias

Reducing or eliminating confounding bias within observational studies is a well studied research topic. In this section, we view predicate-level statistical debugging as an observational study and a novel approach to reduce confounding bias via causal models is presented. While the section describes all of the details needed to understand this approach, a more complete discussion of observational studies and causal models is provided in Section 2.7.

Casting predicate-level statistical debugging as an observational study yields two groups of test case executions. These two execution groups are: those where predicate p is true (the *treatment* group) and those where predicate p is not true (the *control* group) [94]. For a predicate p , the membership of a test case in either the *treatment* or the *control* group is denoted by the treatment variable T_p . For those test cases where predicate p is true, $T_p = 1$. For those test cases where predicate p is not true, $T_p = 0$. Independent of the presence of a given predicate, test case executions can also be classified with an outcome variable Y . A successful test case execution is denoted by $Y=0$ and a failing test case execution is denoted by $Y=1$.

In the context of an observational study, estimating the average treatment effect of a predicate on a test case corresponds to estimating the probability of program Q failing given that a specific predicate p is true,

$\Pr(Q \text{ fails} \mid p=\text{true})$. The average treatment effect of a predicate, τ_p , is estimated by a causal model. The most basic estimate for the average treatment effect of a predicate is computed using the causal model in Equation 4.1. The model is linear and is solved with least-squares regression, α_p is an intercept and σ_p is a random error term that is uncorrelated with T_p [95]. The estimate of $\Pr(Q \text{ fails} \mid p=\text{true})$ computed in this model is the specificity suspiciousness estimate in Section 3.2.1.

1. For each predicate p in a faulty subject program Q , fit a separate linear model M_p according to the following:
 - (a) The *outcome* variable Y is 1 for a test case if it fails and is 0 otherwise.
 - (b) The *treatment* indicator T_p is 1 for a test case if p is true and is 0 otherwise.
2. Rank the predicates in descending order of, $\tau_{ls,p}$. $\tau_{ls,p}$ is the least-squares estimate, for each predicate, of the coefficient of the treatment variable T_p in the causal model M_p .

$$Y = \alpha_p + \tau_p T_p + \epsilon_p \quad (4.1)$$

The causal model shown in Equation 4.1 shows the symmetry between the specificity estimate presented in Section 3.2.1 and the estimate of the average treatment effect of a predicate on subject program test case outcomes. However, the model in Equation 4.1 ignores any dependencies between the treatment variable T_p and the outcome variable Y . This relationship reflects *treatment selection*. Ignoring *treatment selection* is equivalent to assuming the treatment variable T_p and the outcome variable Y are independent. Current suspiciousness estimates, such as the specificity measure, make this assumption. As a result, they suffer from confounding bias.

It is often possible to characterize the process of *treatment selection* in terms of one or more important variables. For example, a physician considers the symptoms, vital signs, and medical history of a patient before selecting a treatment. These variables are *covariates* of the treatment variable T_p . If a set of covariates accounts well for which units in an observational study receive treatment and which do not, then it is possible to reduce or eliminate confounding bias when estimating the average treatment effect.

In the context of statistical debugging, if a set X of covariates accounts well for those test cases where a given predicate p is true ($T=1$) and those test cases where p is not true ($T=0$), then it is possible to reduce or eliminate confounding bias in the suspiciousness estimate for a predicate. This is accomplished by controlling for (or conditioning on) X in a causal model that estimates the average treatment effect of a predicate [94]. In sections 4.2.2 and 4.2.3 causal models which employ this approach to control for different covariates are presented.

4.2.2 Control Flow Dependency Confounding Bias

Our predicate-level adaptation of Baah et al.’s statement-level work for controlling for *control flow dependency* confounding bias begins with defining and identifying the *forward control flow predecessor statement* for a given predicate. This requires a review of control flow graphs and statement dependency.

A program’s control flow graph is a directed graph whose nodes correspond to program statements and whose edges represent control dependences between statements [37]. Node Y is *control dependent* on node X if X has two outgoing edges and the traversal of one edge always leads to the execution of Y while the traversal of the other edge does not necessarily execute Y . Node X *dominates* node Y in a control flow graph if every path from the entry node to Y contains X . Node Y is *forward control dependent* on node X if Y is control dependent on X and Y does not dominate X [37]. Forward control dependences are control dependences that can be realized during execution without necessarily executing the dependent node more than once. Node X is a *forward control flow predecessor* of Node Y if Y is *forward control dependent* on X and X immediately precedes Y in the control flow graph. The statement corresponding to node X is the *forward control flow predecessor statement* of the statement corresponding to node Y that is defined by Baah et al. [31, 32].

We identify the *forward control flow predecessor predicate* for a given predicate p using an approach that is similar to Baah et al.’s approach to identify the *forward control flow predecessor statement* for a given statement $stmt$. Both approaches extract the control-dependence graph of a subject program and identify the *forward control flow predecessor statement* for $stmt$, which corresponds to p , by searching for the control flow statement in the graph immediately preceding $stmt$.

However, identifying the *forward control flow predecessor predicate* for p requires an additional step. Recall, a statement reflects a line of source code while predicates represent conditions that are true about variables within a line of source code. Thus, given a *forward control flow statement*, the *forward control flow predecessor predicate* is located by instrumenting the *forward control flow predecessor statement* with two branch predicates. The first of the two predicates asserts that the branch is false. The second of the two predicates asserts that the branch is true. When the branch is reached, exactly one of the two predicates will be true. The predicate that is true is the *forward control flow predecessor predicate*.

Ultimately, capturing *forward control flow predecessor predicates* requires: (1) static and/or elastic predicates to be instrumented in the subject program, (2) extracting the control flow graph of the subject program and identifying *forward control flow predecessor statements* and (3) the instrumentation of branches within the identified *forward control flow predecessor statements*. If a *forward control flow predecessor statement* cannot be found for $stmt$ then $stmt$ does not have a *forward control flow predecessor statement*.

and the predicate p does not have a *forward control flow predecessor predicate*.

The instrumentation for static, elastic and branch predicates is described in Section 3.3.3. The control flow graph is extracted using the CIL framework, which supports the analysis of ANSI C programs [115]. Only *dynamic* control flow graphs, as opposed to *static* control flow graphs, are searched to identify the *forward control flow predecessor statement* for a given statement $stmt$. Using *dynamic* control-dependence graphs ensures that only control dependences that are exercised at runtime appear in the extracted graph.

The ability to capture the *forward control flow predecessor predicate* for a given predicate enables our causal model which controls for *control flow dependency* confounding bias to be defined. Given an instrumented subject program, and the feedback reports from executing a set of test cases with a predicate-level statistical debugger such as CBI or ESP, the following causal model reduces or eliminates *control flow dependency* confounding bias:

1. For each predicate p in a faulty subject program Q , fit a separate linear model M_p according to the following:
 - (a) The *outcome* variable Y is 1 for a test case if it fails and is 0 otherwise.
 - (b) The *treatment* variable T_p is 1 for a test if p is true and is 0 otherwise.
 - (c) If p has a *forward control flow predecessor predicate*, $cfp(p)$, then M_p has a single binary covariate C_p , which is 1 for a test case if $cfp(p)$ is true and is 0 otherwise. If p does not have a *forward control flow predecessor predicate* then M_p has no covariates.
2. Rank the predicates in descending order of, $\tau_{ls,p}^c$, the least-squares estimate, for each predicate, of the coefficient of T_p in M_p .

The resulting linear model for a predicate p is:

$$Y = \alpha_p^c + \tau_p^c T_p + \beta_p^c C_p + \epsilon_p^c \quad (4.2)$$

The coefficient $\tau_{ls,p}^c$ is the average treatment effect of predicate p on subject test case outcomes. This estimate is a reduced bias version of the specificity measure that is employed in the suspiciousness estimates presented in Section 3.2.

The role of covariate C_p is to control for confounding bias of the suspiciousness estimate for predicate p , due to the truth of other subject program predicates. Intuitively, conditioning on C_p reduces confounding bias because $cfp(p)$ is the most immediate cause of p being evaluated (true or false) or p not being evaluated (true or false) in a particular test case.

Pearls *Back-Door Criterion* for causal graphs, which is extensively covered in Section 2.7, provides a formal justification for the causal model in Equation 4.2. For any subject program where failure is determined with a single output statement and control dependences carry all the causal influences of failure, any back door paths from a predicate to the output statement must begin with the *forward control flow predecessor predicate*, $cfp(p)$. Thus, $cfp(p)$ is a suitable covariate of T_p because it satisfies Pearl's Back-Door Criterion. It blocks all *back-door paths* in the dynamic control flow graph from the predicate corresponding to the treatment variable T_p to output statement corresponding to the outcome variable Y . As a result the *control flow dependency confounding bias* in $\tau_{ls,p}^c$, the model's estimate of the average treatment effect, is reduced.

4.2.3 Failure Flow Confounding Bias

Recall from 3.2.4, Liblit et al.'s *Importance* measure of predicate suspiciousness is very similar to the F_1 measure. Both measures balance an estimate of the probability of a subject program Q failing given that predicate p is true, $\Pr(Q \text{ fails} \mid p=\text{true})$, with the *sensitivity* measure. The difference between Liblit et al.'s *Importance* measure and the F_1 measure lies in the estimate of $\Pr(Q \text{ fails} \mid p=\text{true})$. Liblit et al.'s *Importance* measure estimates $\Pr(Q \text{ fails} \mid p=\text{true})$ with the difference $f_p/(f_p + s_p) - f_{p \text{ obs}}/(f_{p \text{ obs}} + s_{p \text{ obs}})$, while the F_1 measure employs the *specificity* measure.

The difference, $f_p/(f_p + s_p) - f_{p \text{ obs}}/(f_{p \text{ obs}} + s_{p \text{ obs}})$, in the estimate of $\Pr(Q \text{ fails} \mid p=\text{true})$ in Liblit et al.'s *Importance* measure is used as a heuristic to reduce *failure flow confounding bias*. The term measures not the chance that a predicate p implies failure, but how much difference it makes that the predicate p is observed to be true versus simply reaching the line where the predicate p is evaluated [10]. While this difference can be an effective means of reducing *failure flow confounding bias*, it is a heuristic, not a proven solution. Formally reducing or eliminating *failure flow confounding bias* requires a causal model. Furthermore, the set of covariates within the model must satisfy the *Back-Door Criterion*. The difference, $f_p/(f_p + s_p) - f_{p \text{ obs}}/(f_{p \text{ obs}} + s_{p \text{ obs}})$, in Liblit et al.'s *Importance* measure does not meet these requirements. In this subsection, our causal model, which controls for predicate evaluation (true or false), satisfies Pearl's *Back-Door Criterion* and reduces *failure flow* confounding bias, is presented.

Given the outcomes and predicate coverage vectors from executing a set of test cases with a predicate-level statistical debugger such as CBI or ESP the following approach reduces or eliminates both *control flow dependency* and *failure flow* confounding biases:

1. For each predicate p in a faulty subject program Q , fit a separate linear model M_p according to the following:
 - (a) The *outcome* variable Y is 1 for a test case if it fails and is 0 otherwise.

- (b) The *treatment* variable T_p is 1 for a test if p is true and is 0 otherwise.
 - (c) If p has a *forward control flow predecessor predicate*, $cfp(p)$, then M_p has a single binary covariate C_p , which is 1 for a test case if $cfp(p)$ is true and is 0 otherwise. If p does not have a *forward control flow predecessor predicate* then M_p has no covariates.
 - (d) The covariate D_p is 1 for a test case if p is evaluated (true or false) and is 0 if p otherwise.
2. Rank predicates in descending order of $\tau_{ls,p}^{c,f}$, the least-squares estimate, for each predicate, of the coefficient of T_p in M_p .

The resulting linear model for a predicate p is:

$$Y = \alpha_p + \tau_p^{c,f} T_p + \beta_p^{c,f} C_p + \omega_p^{c,f} D_p + \epsilon_p \quad (4.3)$$

Here, the coefficient $\tau_{ls,p}^{c,f}$ is the least-squares estimate of the average treatment effect of a predicate p on the subject program test case outcomes. The notable difference between Equation 4.3 and the causal model shown in Equation 4.2 is the inclusion of the covariate D_p , which reflects whether or not predicate p is evaluated. Intuitively, D_p further reduces confounding bias because it enables the model to determine how much difference it makes that the predicate p is observed to be true versus simply reaching the line where the predicate p is evaluated.

Once again, inclusion of the covariate C_p blocks all back-door paths from the predicate corresponding to the treatment variable T_p to the output statement corresponding to the outcome variable Y . This enables the additional source of bias controlled for by D_p to be reduced without introducing other sources of bias into $\tau_{ls,p}^{c,f}$. This is significant. Without the inclusion of covariate C_p in the model to block all back-door paths, D_p would not be a suitable covariate.

4.2.4 Improving Existing Suspiciousness Estimates

The reduced bias estimates derived in sections 4.2.2 and 4.2.3 can be integrated into the suspiciousness estimates in Section 3.2 to improve their effectiveness. Recall that the measures *Tarantula* and *specificity* are estimates of $\Pr(Q \text{ fails} \mid p=\text{true})$. Similarly, the reduced bias estimates $\tau_{ls,p}^c$ and $\tau_{ls,p}^{c,f}$ are estimates of $\Pr(Q \text{ fails} \mid p=\text{true})$. Thus, integrating either of the reduced bias estimates into the *Tarantula* measure or the *specificity* measure is a straightforward substitution.

However, it is slightly more difficult to integrate $\tau_{ls,p}^c$ and $\tau_{ls,p}^{c,f}$ into the *Ochiai* measure, the F_1 measure and the *Importance* measure. Recall, these metrics balance an estimate of $\Pr(Q \text{ fails} \mid p=\text{true})$ with the *sensitivity* measure. As a result, both $\tau_{ls,p}^c$ and $\tau_{ls,p}^{c,f}$ must be converted into a probability value before

either can replace the biased estimate of $\Pr(Q \text{ fails} \mid p=\text{true})$ in the *Ochiai* measure, the F_1 measure or the *Importance* measure. Keeping with mathematical modeling theory the *inverse logit* function shown in Equation 4.4 is used to convert $\tau_{ls,p}^c$ and $\tau_{ls,p}^{c,f}$ into a probability value [116]. The function constrains the value of the reduced bias estimates to the range 0 - 1 ($[0,1]$) and ensures the combination with the *sensitivity* measure is meaningful.

$$\text{invlogit}(x) = \exp(x)/(1 + \exp(x)) \quad (4.4)$$

Once a reduced bias estimate is converted with Equation 4.4 it can be substituted into the *Ochiai* measure, the F_1 measure or the *Importance* measure in place of the biased estimate of $\Pr(Q \text{ fails} \mid p=\text{true})$. Performing this substitution in the F_1 measure and the *Importance* measure renders the two suspiciousness estimates indistinguishable because they only differ in their estimate of $\Pr(Q \text{ fails} \mid p=\text{true})$. Next, in Section 4.2.5 the effectiveness of each of these reduced bias suspiciousness estimates will be evaluated in CBI and ESP for a large set of subject programs.

4.2.5 Evaluation

In this section an effectiveness evaluation of our reduced confounding bias predicate-level suspiciousness estimates $\tau_{ls,p}^c$ and $\tau_{ls,p}^{c,f}$ is presented. In the evaluation $\tau_{ls,p}^c$ and $\tau_{ls,p}^{c,f}$ are integrated into the *Tarantula*, F_1 , and *Ochiai* suspiciousness estimates and used in the predicate-level statistical debuggers CBI and ESP to localize faults in 28 different subject programs with 328 different, faulty versions.

The 28 subject programs and 328 faulty versions are the same programs used to evaluate the efficiency and effectiveness of the elastic predicates presented in Chapter 3. The subjects, shown in Table 3.4, reflect both established fault localization benchmarks and widely used simulations employing floating-point computations and continuous stochastic distributions. In Table 3.4 the first column gives the program name, the second column provides the ratio of the number of versions used to the number of versions available, the third column gives the number of lines of code for the subject, the fourth column gives the number of test cases and the last column provides a description. For 25 of the faulty versions there were either no syntactic differences between the correct version and the faulty version of the program, or none of the test cases failed when executed on the faulty version of the program. As a result these 25 versions were omitted from the evaluation.

When CBI is employed in the evaluation the subject programs are instrumented with static single variable, static scalar pairs and branches predicates. When ESP is employed subject programs are also instrumented with the FAST ESP elastic single variable and scalar predicates. For CBI and ESP the predicate instrumentation and profiling is implemented using the CIL library. The causal models which yield the

suspiciousness estimates $\tau_{ls,p}^c$ and $\tau_{ls,p}^{c,f}$ for each predicate are implemented using R, which is a statistical computation system with its own language and runtime environment [117].

To study the effectiveness of each suspiciousness estimate, an established cost metric is employed [7, 8, 22, 23, 28, 31, 32]. Given the ranked set of predicates, the metric *Cost* measures the percentage of predicates a developer must examine before encountering the faulty statement. If there are ties, it is assumed that the developer must examine all the tied predicates. For example, if there are n predicates instrumented in a program and all n predicates have the same suspiciousness estimate, it is assumed that the developer must examine all n predicates before discovering the fault. Under this circumstance the *Cost* of localizing the fault is 100%. A lower score is preferable because it means that fewer of the predicates must be considered before the fault is found.

Given a debugger instrumenting subject programs with the same set of predicates, the effectiveness of two suspiciousness estimates A and B is compared by choosing one estimate (B) as the reference metric and subtracting the *Cost* value for the other estimate (A) from the *Cost* of the reference metric (B). If A performs better than B, then the improvement is positive and if B performs better than A, the improvement is negative. For example, for a given program, if the *Cost* of A is 30% and the *Cost* of B is 40%, then the absolute improvement of A over B is 10% because developers would examine, in absolute terms 10% fewer predicates using A instead of B.

For each faulty version of each subject program, the absolute improvement (or lack thereof) of the reduced bias suspiciousness estimate within CBI and ESP is represented with a bi-colored bar. The height of the colored portion of the bar closest to the x-axis reflects the improvement of the estimate for the matching debugger. The total height of both portions reflects the improvement of the metric for the debugger matching the portion of the bar furthest from the x-axis. CBI is shown in red and ESP is shown in white. The effectiveness of CBI relative to ESP is discussed later in this section.

***Tarantula* Suspiciousness Estimate**

Here, the effectiveness of the two reduced biased suspiciousness estimates $\tau_{ls,p}^c$ and $\tau_{ls,p}^{c,f}$ is compared to the standard *Tarantula* suspiciousness estimate. First, the effectiveness of the suspiciousness estimate $\tau_{ls,p}^c$ which controls for control-flow dependency confounding bias is evaluated. In Figure 4.2 the *Cost* of employing $\tau_{ls,p}^c$ is subtracted from the *Cost* of employing the standard *Tarantula* suspiciousness estimate, which serves as the reference metric.

Figure 4.2 shows that both ESP and CBI are more effective at localizing faults in 149 of the 328 faulty versions using the suspicious estimate $\tau_{ls,p}^c$ instead of the standard predicate-level *Tarantula* estimate. In 177 of the 328 faulty versions there is no difference in the effectiveness of ESP and CBI when employing $\tau_{ls,p}^c$ or

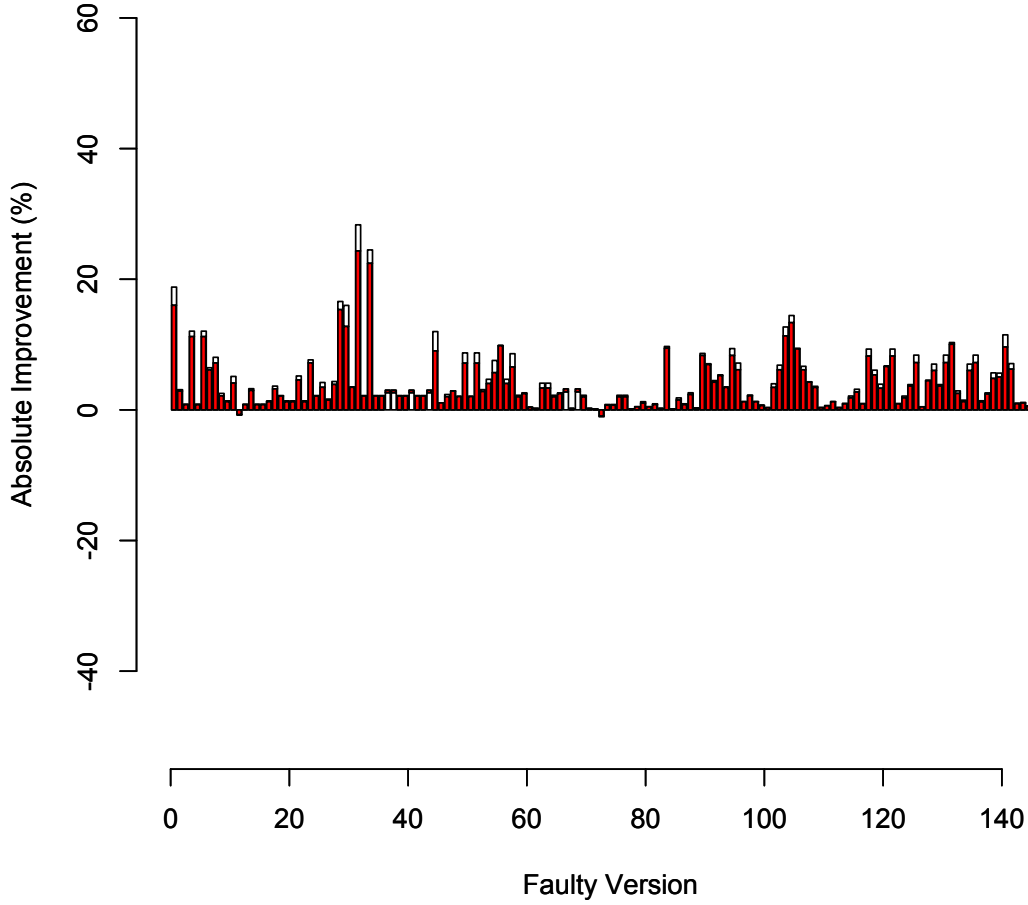


Figure 4.2: Improvements in effectiveness achieved in ESP (red) and CBI (white) by using $\tau_{ls,p}^c$ as the suspiciousness estimate instead of the standard *Tarantula* estimate.

the standard predicate-level *Tarantula* measure. However, in 2 of the 328 versions, CBI and ESP are less effective at localizing faults when employing the suspiciousness estimate $\tau_{ls,p}^c$ as opposed to the standard *Tarantula* estimate. The ineffectiveness of $\tau_{ls,p}^c$ for these 2 faulty versions is discussed in Section 4.2.5.

In Figure 4.3, the effectiveness of the suspiciousness estimate $\tau_{ls,p}^{c,f}$ which controls for both control-flow dependency and failure flow confounding bias is evaluated. However, in this portion of the evaluation the standard *Tarantula* estimate no longer serves as the reference metric, instead the reference metric is $\tau_{ls,p}^c$. Thus the data in Figure 4.3 reflects subtracting the *Cost* of employing $\tau_{ls,p}^{c,f}$ from the *Cost* of employing $\tau_{ls,p}^c$.

Figure 4.3 shows that both ESP and CBI are more effective at localizing faults in 106 of the 328 faulty

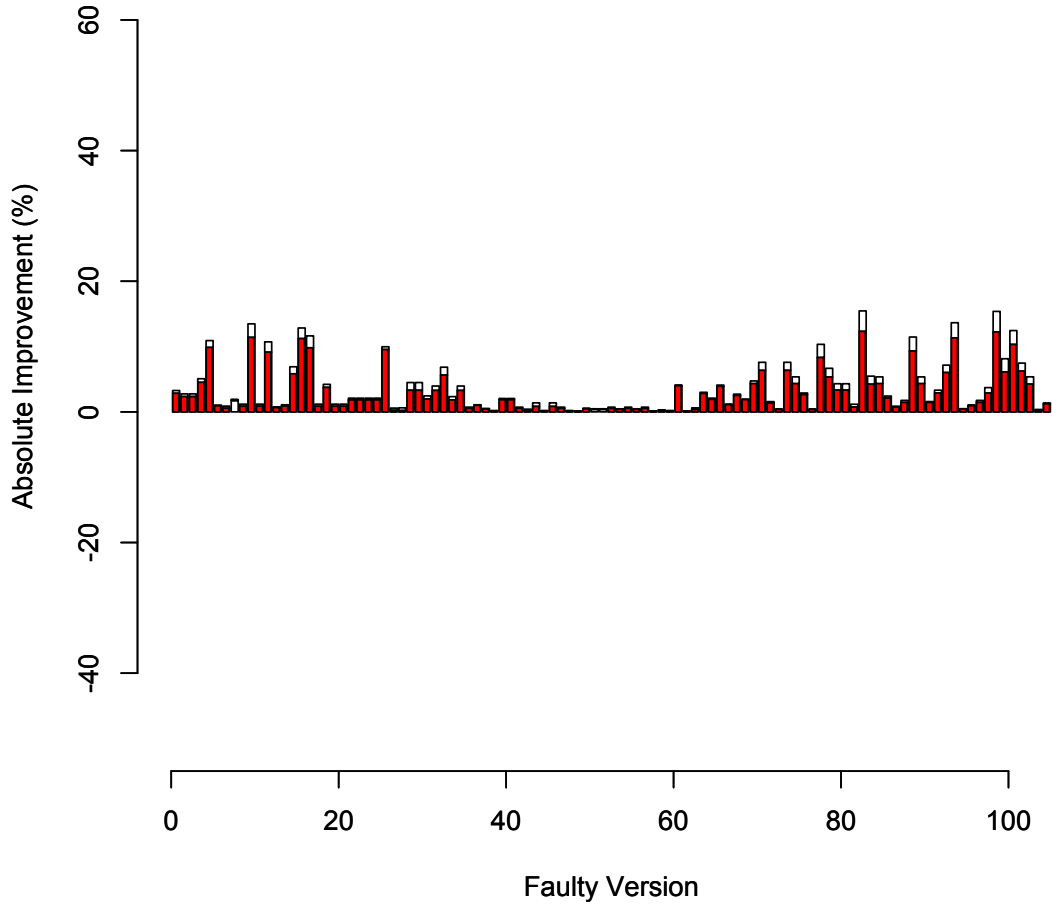


Figure 4.3: Improvements in effectiveness achieved in ESP (red) and CBI (white) by using $\tau_{ls,p}^{c,f}$ as the suspiciousness estimate instead of $\tau_{ls,p}^c$.

versions using the suspicious estimate $\tau_{ls,p}^{c,f}$ compared to the estimate $\tau_{ls,p}^c$. In 222 of the 328 faulty versions there is no difference in the effectiveness of ESP and CBI when employing $\tau_{ls,p}^c$ or $\tau_{ls,p}^{c,f}$. Furthermore, ESP and CBI are never less effective at localizing faults when employing the suspiciousness estimate $\tau_{ls,p}^{c,f}$ compared to the estimate $\tau_{ls,p}^c$.

F_1 Suspiciousness Estimate

Next, the two versions of the F_1 suspiciousness estimate which employ $\tau_{ls,p}^c$ and $\tau_{ls,p}^{c,f}$, respectively, are evaluated. First, the effectiveness of the standard F_1 suspiciousness estimate is compared to the F_1 suspiciousness estimate which uses $\tau_{ls,p}^c$ to estimate $\Pr(Q \text{ fails} \mid p=\text{true})$. In Figure 4.4 the *Cost* of employing the F_1 suspiciousness estimate integrated with $\tau_{ls,p}^c$ is subtracted from the *Cost* of employing the standard F_1 suspiciousness estimate, which serves as the reference metric.

Figure 4.4 shows that both ESP and CBI are more effective at localizing faults in 134 of the 328 faulty versions using the F_1 suspicious estimate integrated with $\tau_{ls,p}^c$ compared to the standard F_1 suspiciousness estimate. In 186 of the 328 faulty versions there is no difference in the effectiveness of ESP and CBI. However, in 8 of the 328 versions, CBI and ESP are less effective at localizing faults when employing the F_1 suspiciousness estimate integrated with $\tau_{ls,p}^c$ as opposed to standard F_1 estimate. The ineffectiveness of the the F_1 suspiciousness estimate integrated with $\tau_{ls,p}^c$ for these 8 faulty versions is discussed later in this section.

In Figure 4.5, the effectiveness of the F_1 suspicious estimate integrated with $\tau_{ls,p}^{c,f}$ which controls for both control-flow dependency and failure flow confounding bias is evaluated. In this portion of the evaluation the standard F_1 suspiciousness estimate no longer serves as the reference metric, instead the reference metric is the F_1 suspicious estimate integrated with $\tau_{ls,p}^c$. Thus the data in Figure 4.5 reflects subtracting the *Cost* of integrating $\tau_{ls,p}^{c,f}$ into the F_1 suspicious estimate from the *Cost* of integrating $\tau_{ls,p}^c$ into the F_1 suspicious estimate.

Figure 4.5 shows that both ESP and CBI are more effective at localizing faults in 98 of the 328 faulty versions when integrating $\tau_{ls,p}^{c,f}$ into the F_1 estimate as opposed to $\tau_{ls,p}^c$. In 230 of the 328 faulty versions there is no difference in the effectiveness of ESP and CBI when integrating $\tau_{ls,p}^c$ or $\tau_{ls,p}^{c,f}$ into the F_1 estimate. Furthermore, ESP and CBI are never less effective at localizing faults when $\tau_{ls,p}^{c,f}$ is integrated into the F_1 estimate instead of $\tau_{ls,p}^c$.

Ochiai Suspiciousness Estimate

The two versions of the *Ochiai* suspiciousness estimate which employ $\tau_{ls,p}^c$ and $\tau_{ls,p}^{c,f}$, respectively, are evaluated next. First, the effectiveness of the standard *Ochiai* suspiciousness estimate is compared to the *Ochiai* suspiciousness estimate which uses $\tau_{ls,p}^c$ to estimate $\Pr(Q \text{ fails} \mid p=\text{true})$. In Figure 4.4 the *Cost* of employing the *Ochiai* suspiciousness estimate integrated with $\tau_{ls,p}^c$ is subtracted from the *Cost* of employing the standard *Ochiai* suspiciousness estimate, which serves as the reference metric.

Figure 4.6 shows that both ESP and CBI are more effective at localizing faults in 80 of the 328 faulty versions using the *Ochiai* suspicious estimate integrated with $\tau_{ls,p}^c$ compared to the standard *Ochiai* suspiciousness

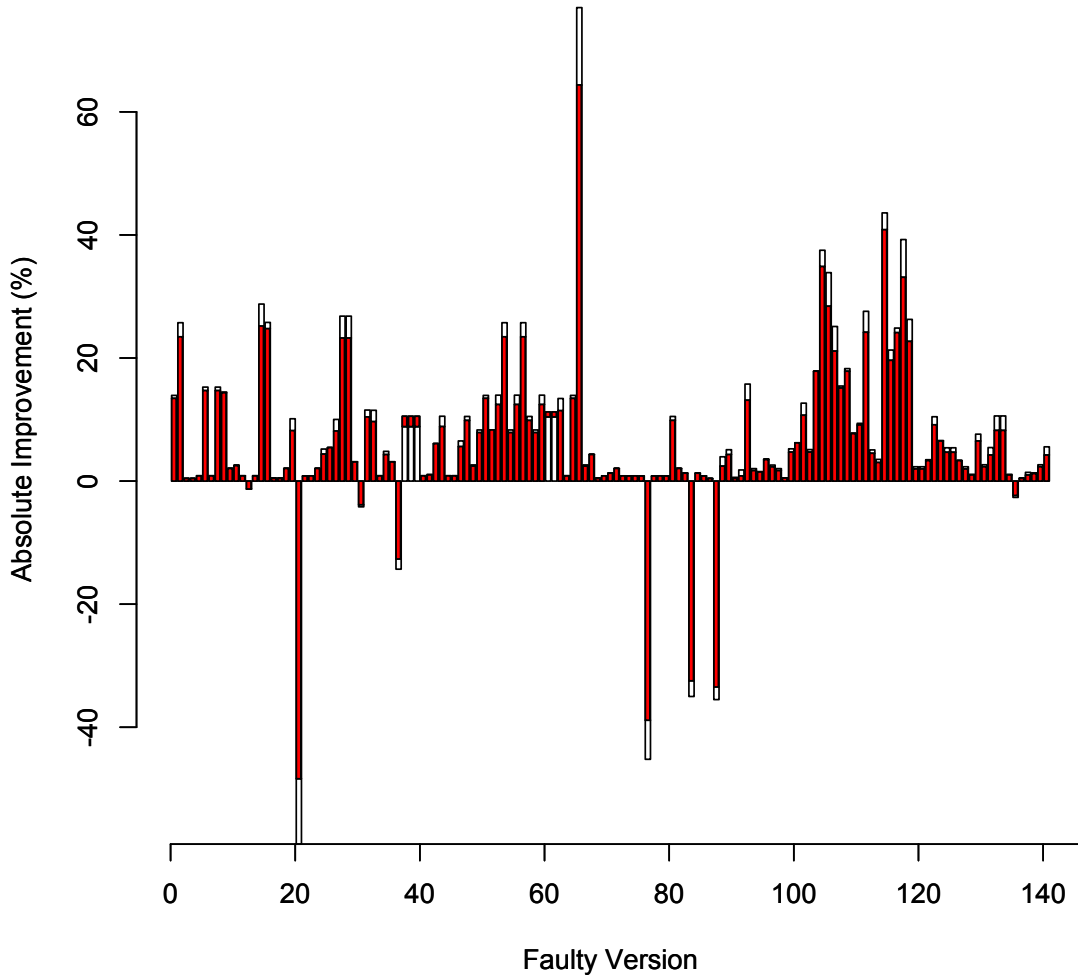


Figure 4.4: Improvements in effectiveness achieved in ESP (red) and CBI (white) by integrating $\tau_{ls,p}^c$ into the F_1 suspiciousness estimate.

estimate. In 245 of the 328 faulty versions there is no difference in the effectiveness of ESP and CBI. However, in 3 of the 328 versions, CBI and ESP are less effective at localizing faults when employing the *Ochiai* suspiciousness estimate integrated with $\tau_{ls,p}^c$ as opposed to standard *Ochiai* estimate.

In Figure 4.7, the effectiveness of the *Ochiai* suspicious estimate integrated with $\tau_{ls,p}^{c,f}$ which controls for both control-flow dependency and failure flow confounding bias is evaluated. However, in this portion of the evaluation the standard *Ochiai* suspiciousness estimate no longer serves as the reference metric, instead the reference metric is the *Ochiai* suspicious estimate integrated with $\tau_{ls,p}^c$. Thus the data in Figure 4.5 reflects subtracting the *Cost* of integrating $\tau_{ls,p}^{c,f}$ into the *Ochiai* suspicious estimate from the *Cost* of integrating $\tau_{ls,p}^c$

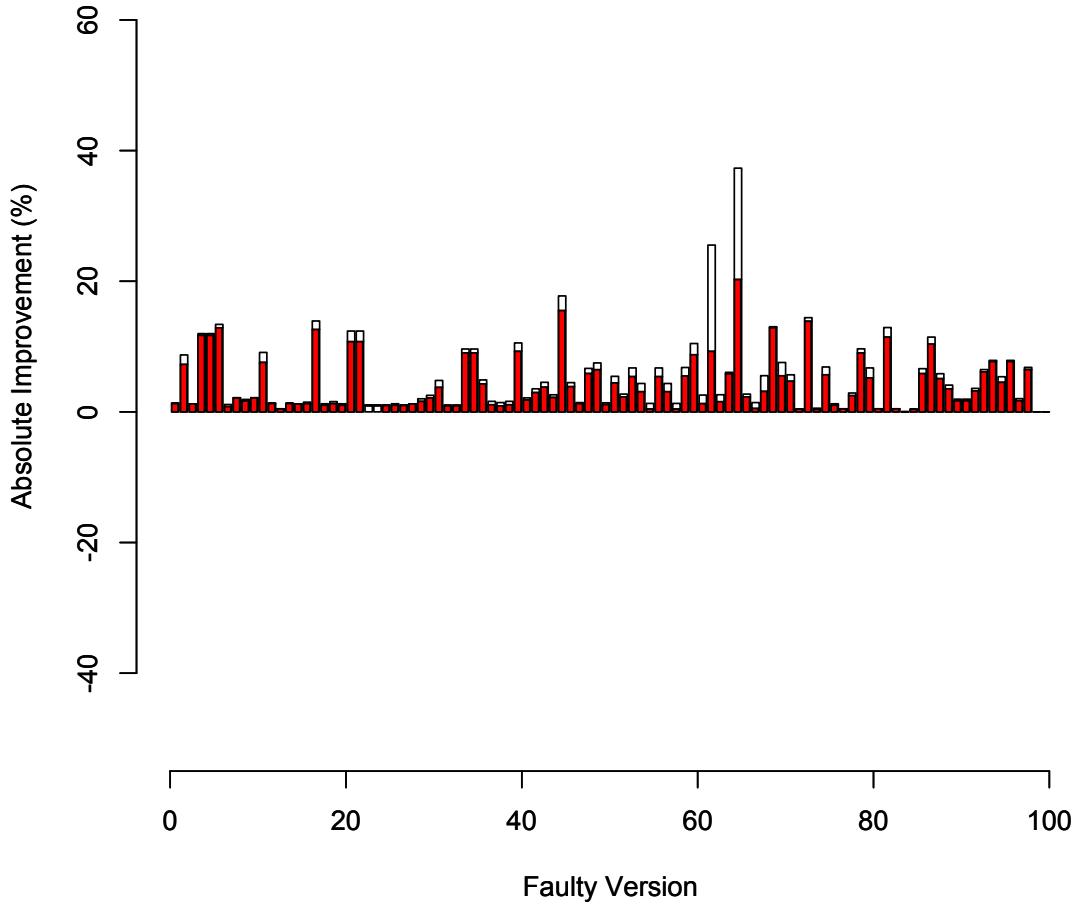


Figure 4.5: Improvements in effectiveness achieved in ESP (red) and CBI (white) by integrating $\tau_{ls,p}^{c,f}$ into the F_1 suspiciousness estimate instead of $\tau_{ls,p}^c$.

into the *Ochiai* suspicious estimate.

Figure 4.7 shows that both ESP and CBI are more effective at localizing faults in 59 of the 328 faulty versions when integrating $\tau_{ls,p}^{c,f}$ into the *Ochiai* estimate as opposed to $\tau_{ls,p}^c$. In 269 of the 328 faulty versions there is no difference in the effectiveness of ESP and CBI when integrating $\tau_{ls,p}^c$ or $\tau_{ls,p}^{c,f}$ into the *Ochiai* estimate. Furthermore, ESP and CBI are never less effective at localizing faults when $\tau_{ls,p}^{c,f}$ is integrated into the *Ochiai* estimate instead of $\tau_{ls,p}^c$.

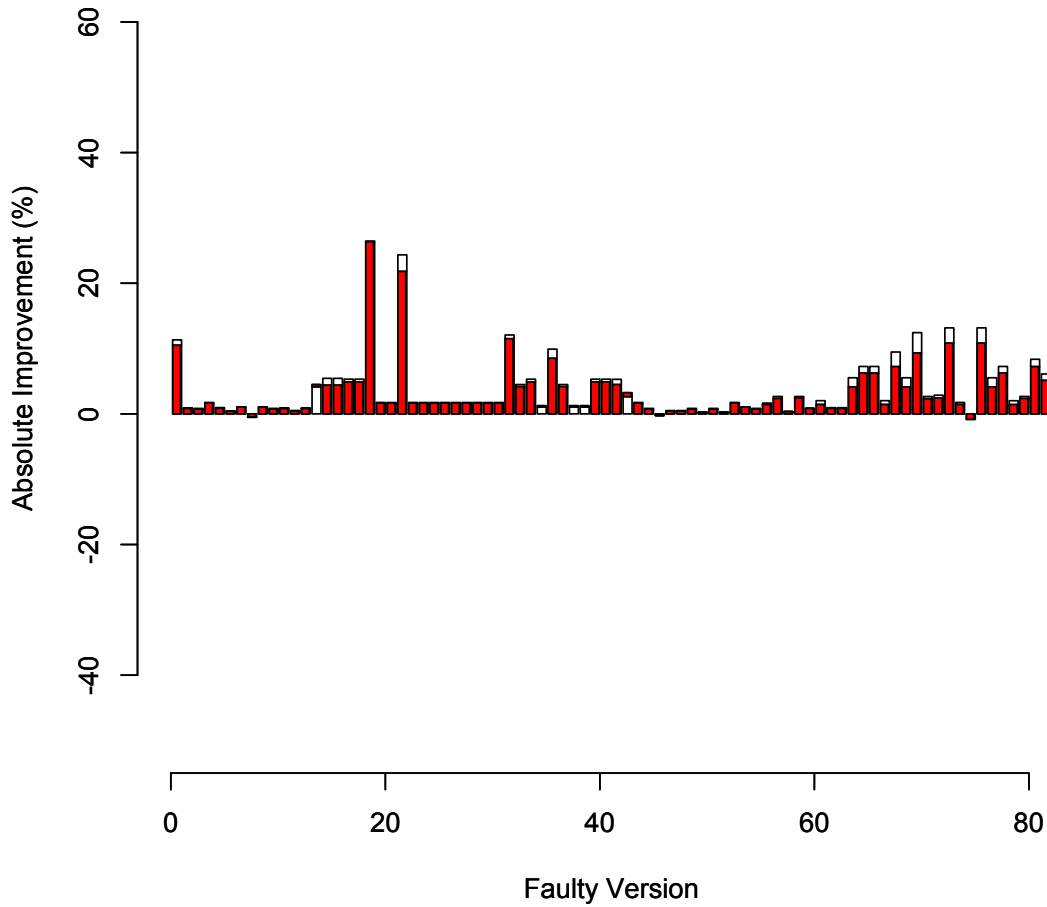


Figure 4.6: Improvements in effectiveness achieved in ESP (red) and CBI (white) by integrating $\tau_{ls,p}^c$ into the *Ochiai* suspiciousness estimate.

Importance Suspiciousness Estimate

In the final portion of the evaluation two different approaches to controlling for failure flow bias are compared: the F_1 suspicious estimate integrated with $\tau_{ls,p}^{c,f}$ and Liblit et al.'s *Importance* estimate. Recall the F_1 suspicious estimate integrated with $\tau_{ls,p}^{c,f}$ formally controls for control flow and failure flow confounding bias via our causal model while the *Importance* estimate employs the difference, $f_p/(f_p + s_p) - f_{p\text{ obs}}/(f_{p\text{ obs}} + s_{p\text{ obs}})$, as a heuristic to factor out failure flow confounding bias. Here, the *Importance* estimate serves as the reference metric and Figure 4.8 reflects subtracting the *Cost* of employing the F_1 suspicious estimate integrated with

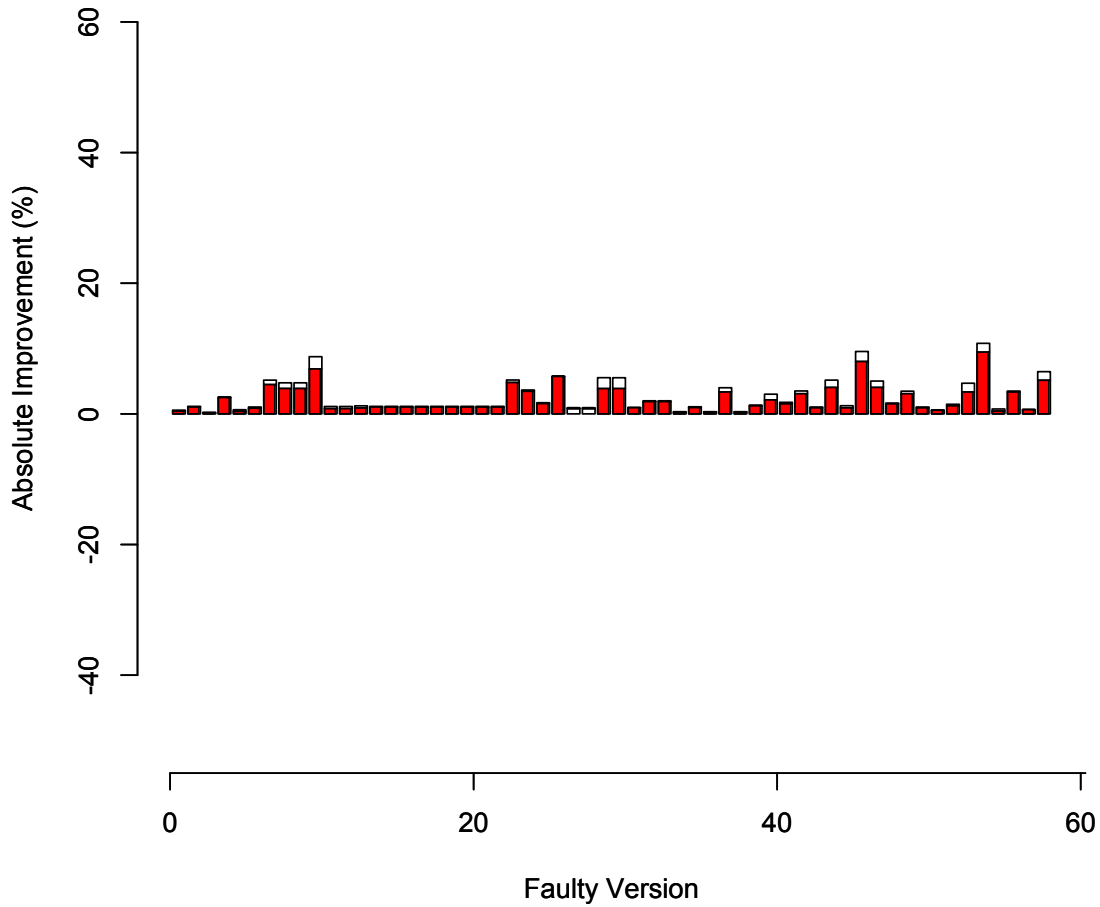


Figure 4.7: Improvements in effectiveness achieved in ESP (red) and CBI (white) by integrating $\tau_{ls,p}^{c,f}$ into the Ochiai suspiciousness estimate instead of $\tau_{ls,p}^c$.

$\tau_{ls,p}^{c,f}$ estimate from the *Cost* of employing Liblit et al.’s *Importance* estimate. Recall from Section 4.2.4 that the F_1 suspicious estimate is used here instead of the *Tarantula* estimate or the *Ochiai* estimate because of its similarity to the *Importance* estimate.

Figure 4.8 shows that both ESP and CBI are more effective at localizing faults in 196 of the 328 faulty versions when controlling for control flow and failure flow confounding bias via our causal model ($\tau_{ls,p}^{c,f}$) instead of employing the difference heuristic in the *Importance* estimate ($(f_p/(f_p + s_p) - f_{p\text{ obs}}/(f_{p\text{ obs}} + s_{p\text{ obs}}))$). In 91 of the 328 faulty versions there is no difference in the effectiveness of ESP and CBI when either estimate

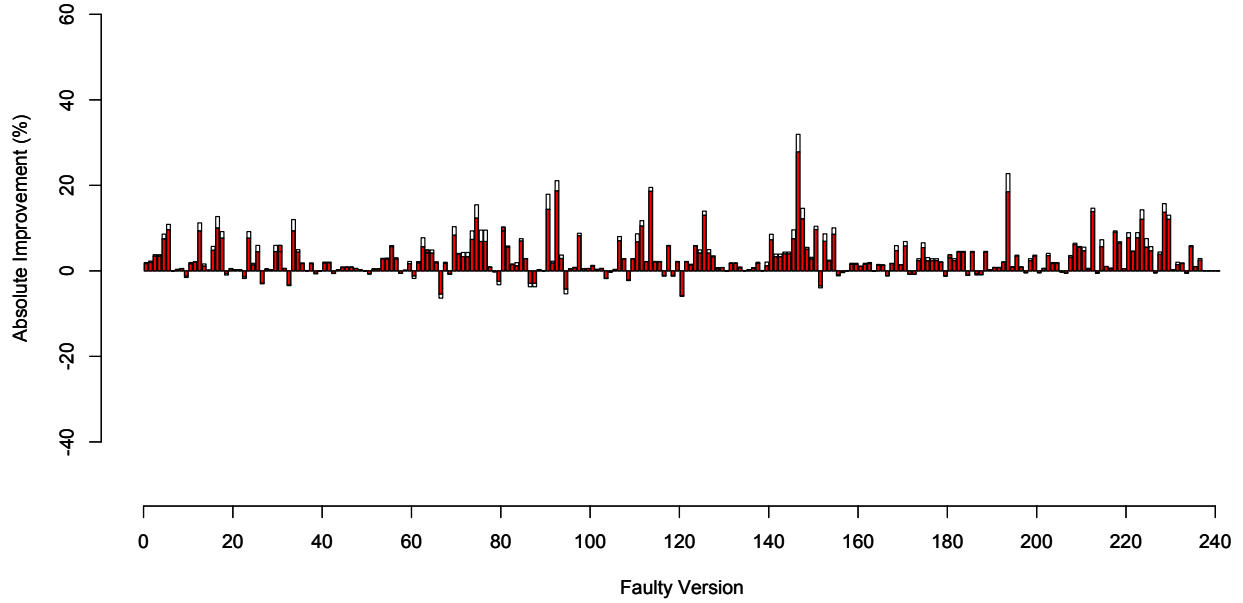


Figure 4.8: Improvements in effectiveness achieved in ESP (red) and CBI (white) by integrating $\tau_{ls,p}^{c,f}$ into the Importance suspiciousness estimate instead of the difference heuristic: $f_p/(f_p + s_p) - f_{p\text{ obs}}/(f_{p\text{ obs}} + s_{p\text{ obs}})$.

is used. However, ESP and CBI are less effective in 41 of the 328 faulty versions at localizing faults when controlling for control flow and failure flow confounding bias via our causal model as opposed to employing Liblit et al.’s *Importance* estimate. The ineffectiveness of our causal model for these 41 faulty versions is discussed later in this section.

Validity

Internal, external, and construct validity threats affect the evaluation. Internal validity threats arise when factors affect the dependent variables without evaluators’ knowledge. It is possible that some implementation flaws in the causal models presented in Section 4.2 could have affected the evaluation results. However, the results for the evaluated benchmarks are similar in magnitude to improvements offered by Baah et al.’s statement-level work [31, 32]. Threats to external validity occur when the results of the evaluation cannot be generalized. Although the evaluation is performed on 28 different subject programs with 328 faulty versions and two different predicate-level statistical debuggers (CBI and ESP), the effectiveness observed in the evaluation cannot be generalized to other faults in other programs for other predicate-level statistical

debuggers. Threats to construct validity concern the appropriateness of the metrics used in our evaluation. More studies into how useful developers find predicate-ranking metrics need to be performed. However, the more accurate fault-localization methods are, the more meaningful such studies will become.

Discussion

In each portion of the evaluation, ESP and CBI are most effective when employing a suspiciousness estimate featuring $\tau_{ls,p}^{c,f}$. These results show that failure and control flow confounding bias exist in the established suspiciousness estimates presented in Section 3.2 and our causal model presented in Section 4.2.3 yields a suspicious estimate ($\tau_{ls,p}^{c,f}$) that reduces or eliminates these biases.

Although the suspiciousness estimates $\tau_{ls,p}^c$ and $\tau_{ls,p}^{c,f}$ performed well in each portion of the evaluation, they were not as effective as their standard measure counterparts for a small number of faulty subject program versions. The faults in these versions violate the *coverage trigger assumption*, which assumes that the coverage of a statement corresponding to predicate p will necessarily trigger a failure, if the statement is faulty [31, 32]. However, covering a faulty statement corresponding to predicate p may not be sufficient to trigger a failure because either the statement does not cause an invalid internal state or an invalid internal state does not propagate to the program's output. Often these faults correspond to missing statements where the predicates corresponding to statements adjacent to the missing code qualify as the fault.

The effectiveness of $\tau_{ls,p}^c$ and $\tau_{ls,p}^{c,f}$ relative to CBI and ESP is also important to discuss. For the preponderance of the program versions CBI offers more improvement than ESP. However, for most of the program versions ESP incurs less overall *Cost* for developers. This paradox can be explained. Recall from Chapter 3 that ESP has been shown to be more effective than CBI when standard suspiciousness metrics are employed [26]. While $\tau_{ls,p}^c$ and $\tau_{ls,p}^{c,f}$ improve the effectiveness of each predicate-level debugger, ESP appears to improve less by absolute measure because of its superior effectiveness. Similarly, for most of the versions where negative improvement is observed, the effectiveness of ESP degrades less than CBI.

Although the suspiciousness estimate $\tau_{ls,p}^{c,f}$ performed well in each portion of the evaluation there are still faulty versions where its effectiveness can be improved. For example, the $\tau_{ls,p}^{c,f}$ integrated version of the F_1 estimate is shown to be less effective than Liblit et al.'s *Importance* estimate in 41 different faulty versions of the subject programs. While several of these versions violate the *coverage trigger assumption*, all of them do not, suggesting that limitations in our causal model exist.

One limitation of the model is that it does not fully account for the composition of the test suite used to compute the suspiciousness estimate $\tau_{ls,p}^{c,f}$ for a given predicate p . Test suite composition can be problematic for p , when the set of tests where p is true and the set of tests where p is *not* true differ substantially with respect to control flow dependency and statement coverage. Under these circumstances, the suspiciousness

estimate for p yielded from our causal model ($\tau_{ls,p}^{c,f}$) is unreliable. In the next section a strategy to address the composition of test suites for a given predicate is presented.

4.3 Matching

Recall, our causal model presented in Section 4.2.3 reduces confounding bias in the suspiciousness estimate for a predicate p by considering two groups of test cases: those test cases where p is true (the *treatment* group) and those test cases where p is not true (the *control* group). In order to produce reliable suspiciousness estimates in the model, the test cases where p is true should have the same pattern of control flow dependencies and predicate evaluations as the test cases where p is not true. When this condition is met, the groups are balanced. When it is not met, there is a *lack-of-balance* and suspiciousness estimates from the causal model using these test cases become unreliable [94, 95].

In practice, the set of subject program test cases used in predicate-level statistical debugging is given and the test cases in the *treatment* and *control* group cannot be assumed to be balanced. Furthermore, generating a random test suite with respect to a set of predicates is a non-trivial task that is made even more computationally expensive when elastic predicates are employed [32].

In this section, instead of generating such a test suite, test cases within the existing test suite for a subject program are excluded to create a set of test cases that mitigates *lack-of-balance* issues for each predicate. To achieve relative balance between the *treatment* and *control* groups and thus create a more reliable suspiciousness estimate, a statistical technique called *matching* is employed. For each predicate p , the test cases in the two groups are reorganized so that they are similar with respect to control flow dependencies and predicate evaluations for p . Each test case tc where p is true is matched with the test case that is most similar to tc where p is not true.

After the test cases are matched, the suspiciousness estimate for p is computed using the causal model shown in Equation 4.1, which does not include any covariate terms. This model is used instead of the model which includes covariates in Equation 4.3 because the matching process controls for the confounding bias, making the inclusion of additional covariates unnecessary [94, 95].

4.3.1 Motivating Example

An example helps elucidate how the *lack-of-balance* in a subject program test suite can create unreliable predicate suspiciousness estimates. The source code shown in Figure 4.9 is taken from an implementation of the Traffic Collision Avoidance System (TCAS) available at the Software-artifact Infrastructure Repository

```

124 if (enabled && ((tcas_equipped && intent_not_known)) // BUG: || !tcas_equipped))
125 {
126     need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
127     need_downward_RA = Non_Crossing_Biased_Descend() && Own_Above_Threat();
128     if (need_upward_RA && need_downward_RA)
129         alt_sep = UNRESOLVED;
130     else if (need_upward_RA)
131         alt_sep = UPWARD_RA;
132     else if (need_downward_RA)
133         alt_sep = DOWNWARD_RA;
134     else
135         alt_sep = UNRESOLVED;
136 }

```

Figure 4.9: An excerpt from a faulty implementation of `tcas`.Table 4.3: Execution data for predicate $(\text{need_upward_RA} > 0)_{126}$ in Figure 4.9.

# of Tests	Predicate	Forward Control Flow Predecessor	Predicate Evaluated	Failure
294	1	1	1	1
721	1	1	1	0
40	0	1	1	0
516	0	0	0	0

(SIR) [118]. TCAS monitors an aircraft’s airspace and warns pilots of possible collisions with other aircraft [119]. Statement 124 contains a fault, which causes some subject program test cases to fail.

Table 4.3 summarizes execution data gathered for the static predicate $(\text{need_upward_RA} > 0)_{126}$, in Figure 4.9. An entry of ‘1’ in Table 4.3 denotes that the characteristic was observed in the specified number of execution traces and an entry of ‘0’ denotes the characteristic was not observed in the specified number of execution traces. Combinations of characteristics not listed in Table 4.3 do not occur for the TCAS test suite.

The test cases shown in Table 4.3 for the predicate $(\text{need_upward_RA} > 0)_{126}$ are not balanced. In all of the tests where $(\text{need_upward_RA} > 0)_{126}$ is true (*treatment* group), the *forward control flow predecessor predicate*, $\text{cfp}(\text{need_upward_RA} > 0)_{126}$, for the predicate is also true and $(\text{need_upward_RA} > 0)_{126}$ is evaluated.

However, in only 40 of the tests where predicate $(\text{need_upward_RA} > 0)_{126}$ is not true (*control* group), is the control flow predecessor for p , $\text{cfp}(\text{need_upward_RA} > 0)_{126}$, true. The remaining 516 tests branch away before reaching $\text{cfp}(\text{need_upward_RA} > 0)_{126}$ or Statement 126. These differences represent a *lack-of-balance* in the test cases for $(\text{need_upward_RA} > 0)_{126}$.

It is important to note that in Table 4.3 two types of *control* group test cases are shown: (1) test cases where the $\text{cfp}(p)$ is true and p is evaluated and (2) test cases where the $\text{cfp}(p)$ is not true and p is not evaluated. However, for some programs a third type of test case is possible in the control group: one where the $\text{cfp}(p)$ is true but p is not evaluated. Test cases with these characteristics correspond to program executions

that fail (crash) after the $cfp(p)$ is evaluated and found to be true, but before p is reached and evaluated. These test cases are discussed further in Section 4.3.3.

While three types of test cases are possible in the *control* group, only test cases where the $cfp(p)$ is true and p is evaluated can be included in the *treatment* group. This restriction exists because the *treatment* group requires that p is true which implies p is evaluated and the $cfp(p)$ is true. In the next section the undesirable results of estimating the suspiciousness of a predicate with unbalanced test cases is explored and *exact matching* is introduced to provide balance.

4.3.2 Exact Matching

In *exact matching*, each *treatment* group unit is matched with one or more *control* group units that have exactly the same covariate values [94]. In the context of predicate-level statistical debugging, *exact matching* excludes tests where predicate p is not true and the predicate p is not evaluated when the suspiciousness of p is estimated. The only tests from the control group that are used for estimation are those where $cfp(p)$ is true and predicate p is evaluated. This exclusion ensures that the resulting control and treatment groups are balanced with respect to: (1) control flow dependencies for a predicate p and (2) the evaluation of a predicate p . This property holds because each of these covariates will be true in all the test cases in both groups.

An example helps elucidate the purpose of *exact matching*. Consider the predicate $(\text{need_upward_RA} > 0)_{126}$. *Exact matching* excludes 516 test cases from the test suite that is used to fit the causal model for this predicate. These tests are reflected in the bottom row of Table 4.3. If matching is not employed when estimating the suspiciousness of $(\text{need_upward_RA} > 0)_{126}$, then the predicate suspiciousness estimate for the predicate is 0.29. However, if the 516 unmatched test cases are excluded, the predicate suspiciousness estimate becomes 0.24. Furthermore, the suspiciousness of the predicate $(\text{enabled} \ \&\& \ ((\text{tcas_equipped} \ \&\& \ \text{intent_not_known}) \ || \ \text{tcas_equipped})) = \text{true})$ in Statement 124 is 0.27 whether or not matching is employed. Thus without *exact matching* the predicate $(\text{enabled} \ \&\& \ ((\text{tcas_equipped} \ \&\& \ \text{intent_not_known}) \ || \ \text{tcas_equipped})) = \text{true})$ in Statement 124 will appear less suspicious than the predicate $(\text{need_upward_RA} > 0)_{126}$. This is incorrect. The fault in Figure 4.9 lies in Statement 124. By excluding the unmatched test cases we are able to more accurately estimate the suspiciousness of the predicates in Figure 4.9 and more effectively identify the fault in this implementation of TCAS.

Unfortunately, the stringent requirements of *exact matching* can result in a complete *lack of overlap* between test cases in the treatment and control group. When this occurs there is no basis to estimate suspiciousness of a predicate. Within the context of predicate-level statistical debugging a *complete lack of*

overlap occurs for a given predicate p when there are no test cases where the $cfp(p)$ of p is true, p is evaluated and p is not true.

We estimate the suspiciousness of a predicate p with a complete lack of overlap by imputing to p the suspiciousness of the control flow ancestor of p that does not have a *complete lack of overlap* problem. This approach is called *suspiciousness imputation* and is shown in Algorithm 1. The algorithm takes as input a predicate p and returns the imputed suspiciousness. To impute suspiciousness, the algorithm first retrieves the matched test cases for p at Line 1. At Line 2, the algorithm checks whether the set of test cases is empty, signifying a *complete lack of overlap*. If so, the algorithm is called recursively with the control flow predecessor predicate, $cfp(p)$, as the argument. The algorithm terminates when an ancestor of p is found that does not have a *complete lack of overlap* problem.

Algorithm 1 *Suspiciousness imputation* for predicates with a *complete lack of overlap* problem.

```

IMPUTESUSP( $p$ )
1   $matchedTestCases \leftarrow \text{GETMATCHEDTESTCASES}(p)$ 
2  if ( $matchedTestCases = \emptyset$ )
3    then
4       $ancestor \leftarrow \text{GETCFP}(p)$ 
5       $suspiciousness \leftarrow \text{IMPUTESUSP}(ancestor)$ 
6  // Fit Equation 4.1 with matched test cases.
7   $suspiciousness \leftarrow \tau_{ls,p}$ 
8  return  $suspiciousness$ 

```

In the next subsection a more advanced matching technique is presented. The technique, *Mahalanobis Distance (MD) matching*, has less stringent requirements than *exact matching* and its application results in fewer predicates with a *complete lack of overlap* problem.

4.3.3 Mahalanobis Distance Matching

To obtain more flexibility in matching *treatment* group and *control* group units, more advanced matching techniques, such as *Mahalanobis Distance (MD) matching* exist [120]. The *MD* metric, $d_M(a, b)$, measures the similarity between two vectors. Equation 4.5 shows the computation of $d_M(a, b)$ between two vectors a and b where T is the vector transpose and S^{-1} is the inverse of the covariance matrix for a and b .

$$d_M(a, b) = \sqrt{(a, b)^T S^{-1} (a - b)} \quad (4.5)$$

In the context of predicate-level statistical debugging, a reflects one covariate vector from a test case where predicate p is true (*treatment* group) and b reflects one covariate vector from a test case where predicate

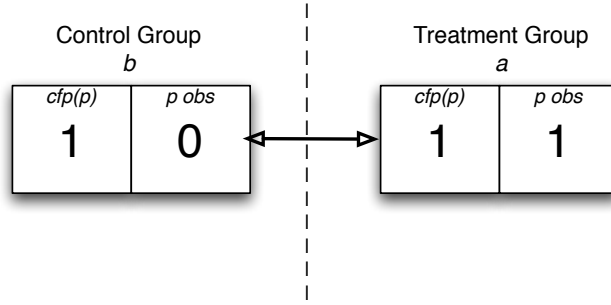


Figure 4.10: The pairing of test cases in *MD matching* between control and treatment groups which reduces *complete lack of overlap*.

p is not true (*control* group). The matrix S is the sample covariance matrix for all of the test cases. In *MD matching*, a treatment unit a is matched with a control unit b if and only if $d_M(a, b)$ is minimal.

Recall, some instrumented predicates do not have a *control flow predecessor predicate*. For these predicates the length of covariate vectors a and b is one, and the results of employing *MD matching* and *exact matching* are identical. However, for predicates with a *control flow predecessor predicate*, the length of a and b is two and employing *MD matching* can avoid a *complete lack of overlap* problem. Specifically, employing *MD matching* avoids a *complete lack of overlap* problem for a predicate p with control group test cases where the *control flow predecessor predicate* ($cfp(p)$) is true but the predicate p is not evaluated.

Recall, these test cases correspond to program executions that fail (crash) after the $cfp(p)$ is found to be true but before p can be evaluated. In *MD matching* these test cases are matched with test cases in the *treatment* group where the $cfp(p)$ is true and p is evaluated. This is shown in Figure 4.10. Intuitively, matching these *treatment* and *control* group test cases together makes sense. These *control* group test cases are failing executions where p would have been evaluated if the program had failed gracefully. Furthermore, because the program failed and p was not evaluated (or true), the inclusion of these *control* group test cases correctly reduces the suspiciousness estimate for p .

Algorithm 2 shows a straight forward implementation of *MD matching* to balance *control* and *treatment* group test cases. Within Algorithm 2, Tr is the set containing the test cases and execution profiles for those test cases where predicate p is true, Con is the set containing the test cases and execution profiles for those test cases where predicate p is not true. The algorithm returns the balanced set of test cases, $Tests$, which is used to fit the causal model specified in Equation 4.1 and compute the suspiciousness estimate for a given predicate.

If the distance between any test case in the control group and any test case in the treatment group is finite then a nonempty set of tests is returned via $Tests$. However, if the $d_M(a, b)$ between every test case

in the control group and every test case in the treatment group is infinite, then a *complete lack of overlap* problem for the predicate cannot be avoided and Algorithm 1 must be employed to impute the suspiciousness of the predicate from a control flow ancestor.

In the next section the decreases in efficiency and improvements in effectiveness that result from employing *exact matching* and *MD matching* to identify test cases to estimate the suspiciousness of a predicate p are explored.

Algorithm 2 *MD matching* to identify test cases to estimate the suspiciousness of a predicate.

```

GETMATCHEDTESTCASES( $Tr, Con$ )
1   $Tests \leftarrow \emptyset$ 
2   $index \leftarrow 0$ 
3  for (each  $Tr_i \in Tr$ )
4      do
5           $min \leftarrow \infty$ 
6          for (each  $Con_j \in Con$ )
7              do
8                   $distance \leftarrow d_M(Tr_i, Con_j)$ 
9                  if ( $distance < min$ )
10                     then
11                          $min \leftarrow distance$ 
12                          $index \leftarrow j$ 
13             if ( $min \neq \infty$ )
14                 then
15                     add  $Con_{index}$  to  $Tests$ 
16                     remove  $Con_{index}$  from  $Con$ 
17 if ( $Tests \neq \emptyset$ )
18     then
19         add  $Tr$  to  $Tests$ 
20 return  $Tests$ 

```

4.4 Evaluation

This section evaluates the effectiveness and efficiency of employing *exact matching* and *MD matching* to identify test cases to estimate the suspiciousness of instrumented predicates in ESP and CBI. The subject programs included in the evaluation are the same 28 subject programs and 328 faulty versions used to evaluate the effectiveness of the causal models presented in Section 4.2.5 and the elastic predicates presented in Chapter 3. They are described in Section 3.4.1 and shown in Table 3.4.

CBI instruments each faulty version of the subject programs with static *single variable*, static *scalar pairs* and branches predicates, while ESP also instruments the subject program with FAST ESP elastic *single variable* and elastic *scalar pairs* predicates. For CBI and ESP the predicate instrumentation and profiling is

implemented using the CIL library [115]. The *exact matching* and *MD matching* algorithms to identify the matched set of test cases for each instrumented predicate are implemented in R [117, 121]. The causal model in Equation 4.1 yields the suspiciousness estimate for each predicate and is also implemented in R [117].

Recall, that given our covariates there is only one difference between *MD matching* and *exact matching*. This difference lies in the restrictions on test cases in the *control* group that can be matched with test cases in the *treatment* group. *MD Matching* matches all *control* group test cases where the *control flow predecessor predicate* is true. However, *exact matching* only matches *control* group test cases where both the *control flow predecessor predicate* is true and the predicate is evaluated.

This enables us to implement *MD matching* as a less restrictive form of *exact matching* where the $d_M(a, b)$ metric, which entails matrix inversion, does not need to be computed. Avoiding the use of the $d_M(a, b)$ metric enables this implementation of *MD matching* to remain efficient compared to *exact matching*. The efficiency of the two matching approaches is discussed further in Section 4.4.

Effectiveness

To measure the effectiveness of the test case matching techniques included in the evaluation, the cost-measuring function *Cost* from Section 3.4 is employed. Recall, *Cost* measures the percentage of predicates a developer must examine before the fault is found, assuming the predicates are sorted in descending order of suspiciousness. To compare two statistical debuggers A vs. B in terms of effectiveness, the *Cost* value for A is subtracted from the *Cost* value for B. The positive difference of A from B reflects the percentage of fewer predicates a developer must examine when employing debugger A as opposed to debugger B. The negative difference reflects the percentage of additional predicates a developer must examine when employing debugger A instead of debugger B.

Tables 4.4 and 4.5 compare the *Cost* of employing *exact matching*, *MD matching*, and not matching test cases to estimate the suspiciousness of instrumented predicates in CBI and ESP for the subject programs. The versions of CBI and ESP employing *exact matching* are labeled EM, the versions employing *MD matching* are labeled MDM and the versions not employing matching are labeled NM. Recall, the NM versions of CBI and ESP estimate the suspiciousness of predicates using the causal model with covariates in Equation 4.3 while EM and MDM use the causal model without covariates in Equation 4.1.

The first column of each table shows the two techniques being compared. The second column (**Positive %**) shows the percentage of faulty versions where difference in *Cost* is positive, the third column (**Neutral %**) shows the percentage of faulty versions where there is no difference in *Cost* and the fourth column (**Negative %**) shows the percentage of faulty versions where the difference in *Cost* is negative. Table 4.4 and Table 4.5 also show the minimum (**Min**), mean (**Mean**) and maximum (**Max**) improvement or degradation for the

Table 4.4: Comparison of matching techniques within CBI.

Comparing	Positive %			Neutral %	Negative %		
	Min	Mean	Max		Min	Mean	Max
MDM vs NM	30.49			67.08	2.44		
	0.03	5.61	15.90		0.03	2.64	9.08
EM vs NM	23.78			73.18	3.05		
	0.03	4.65	10.21		0.03	3.06	10.32
MDM vs EM	6.71			92.68	0.61		
	0.02	1.84	5.69		0.02	0.18	1.24

Table 4.5: Comparison of matching techniques within ESP.

Comparing	Positive %			Neutral %	Negative %		
	Min	Mean	Max		Min	Mean	Max
MDM vs NM	31.71			66.49	1.83		
	0.02	4.27	12.25		0.03	1.76	6.12
EM vs NM	22.86			74.70	2.44		
	0.02	3.26	8.19		0.02	2.14	6.84
MDM vs EM	8.85			90.54	0.61		
	0.01	1.14	5.05		0.01	0.21	0.72

Positive % Column and **Negative %** Column. For example in Table 4.4 comparing MDM vs NM, for the 30.49% of faulty versions where MDM outperforms NM, the minimum *Cost* improvement was 0.03% fewer predicates, the mean *Cost* improvement was 5.61% fewer predicates and maximum *Cost* improvement was 15.90% fewer predicates.

Figure 4.11 summarizes the *Cost* of the ESP and CBI versions of MDM compared to the respective versions of NM over all versions of the subject programs. For each subject program version, the absolute improvement (or degradation) in *Cost* provided by the ESP and CBI versions of MDM is represented with a bi-colored bar. ESP is shown in red and CBI is shown in white. The height of the colored portion of the bar closest to the x-axis reflects the improvement for the matching debugger. The total height of both portions reflects the improvement for the debugger matching the colored portion of the bar furthest from the x-axis. For example, for the first subject program version in the graph, using *MD matching* in ESP resulted in examining 6.31% fewer predicates than not using matching in ESP. For the same version on the graph, using *MD matching* in CBI resulted in examining 7.56% fewer predicates than not using matching in CBI.

Overall Figure 4.11 shows that MDM performed better than NM for both CBI and ESP. Table 4.4 shows that within CBI, MDM performed better than NM on 22.86% of the faulty versions, worse on 2.43% of the faulty versions, and did not change the *Cost* of localizing the fault in 67.08% of the versions. Table 4.5 shows that within ESP, MDM performed better than NM on 31.71% of the faulty versions, worse on 1.83% of the faulty versions, and did not change the *Cost* of localizing the fault in 66.49% of the versions. Furthermore, the improvements in the *Cost* incurred by ESP and CBI with MDM are significantly greater

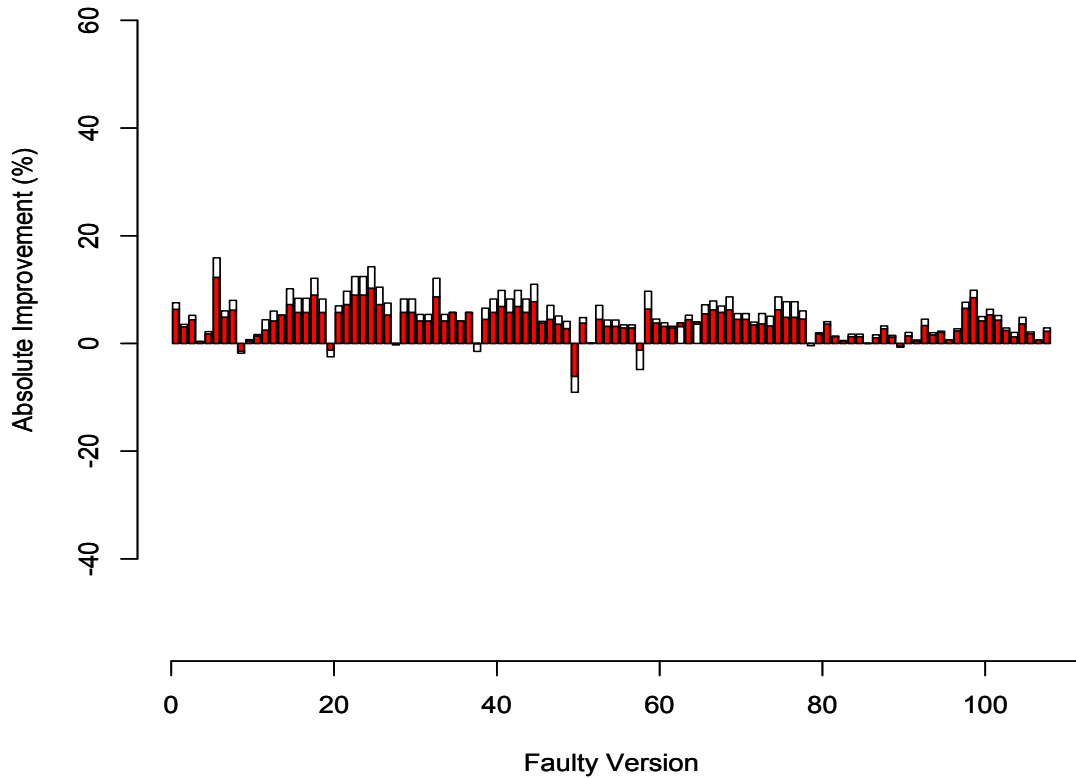


Figure 4.11: Effectiveness of MDM vs NM in ESP (red) and CBI (white).

than the degradations.

EM also performed better than NM for both CBI and ESP in the evaluation. Table 4.4 shows that within CBI, EM performed better than NM on 23.78% of the faulty versions, worse on 3.05% of the faulty versions, and did not change the *Cost* of localizing the fault in 73.18% of the versions. Table 4.5 shows that within ESP, EM performed better than NM on 22.86% of the faulty versions, worse on 2.44% of the faulty versions, and did not change the *Cost* of localizing the fault in 74.70% of the versions. Similarly to MDM, the improvements in *Cost* incurred by ESP and CBI with EM are significantly greater than the degradations.

Although MDM performed better than EM for both ESP and CBI, it is important to explore how much of the improvement resulted from the choice to use *MD matching* as opposed to *exact matching*. To do this MDM is directly compared to EM in the third row of tables 4.4 and 4.5. Recall, in the context of this evaluation the difference between these two matching techniques is the capability of *MD matching* to pair

test cases in the *control* group where the $cfp(p)$ is true but p is not evaluated, with test cases in the *treatment* group. When *exact matching* is employed these *control* group test cases are excluded and a *complete lack of overlap* between the *control* and *treatment* groups can result.

Table 4.4 shows that within CBI, MDM performed better than EM on 6.71% of the faulty versions and only performs worse in 0.61% of the versions. Similarly in Table 4.5 for ESP, MDM performed better than EM on 8.85% of the versions and only performs worse in 0.61% of the versions. Furthermore, for each of the subject program versions where an improvement in *Cost* is observed, the program version crashes instead of failing gracefully. These are exactly the faulty versions that the less restrictive requirements of *MD matching* address. Given this data it is clear that matching the *control* group test cases where the $cfp(p)$ is true but p is not evaluated with test cases in the *treatment* group yields more effective predicate suspiciousness estimates.

Efficiency

In this section the efficiency of the versions of CBI and ESP employing NM and MDM are analyzed. Recall, NM reflects using the causal model in Equation 4.3 which controls for *control flow* and *failure flow* confounding bias, but does not employ matching, to estimate the suspiciousness of predicates. Table 4.6 shows the wallclock time for ranking the predicates for each faulty version of the subject programs using the versions of CBI and ESP employing NM and MDM. Figure 4.12 shows the data in Table 4.6 graphically.

Instead of measuring efficiency in terms of wallclock time, Figure 4.12 measures efficiency in terms of *Slowdown*. Recall, *Slowdown* is a relative efficiency measure which compares the wallclock time of an approach for all versions of a subject program to the wallclock time of a faster, reference approach for the same versions of the same subject program. The formula for *Slowdown* is shown in Equation 4.6. The original versions of CBI and ESP presented in Chapter 3 serve as the faster, reference approach for the NM and MDM versions of ESP and CBI in Figure 4.12.

$$Slowdown = \frac{Runtime_{NM/MDM}}{Runtime_{Original}} \quad (4.6)$$

The efficiency data for EM is not included in Table 4.6 and Figure 4.12 because it is extremely similar to the efficiency data for MDM. This similarity is due to the almost identical implementations of EM and MDM in CBI and ESP. Recall, given our covariates *MD matching* can be implemented as a less restrictive form of *exact matching* where the $d_M(a, b)$ metric and matrix inversion usually entailed in *MD matching* is not required. As a result, the largest discrepancy in wallclock time between the two approaches over all of the versions of any of the subject programs is less than one second, compared to total execution times in the hundreds and thousands of seconds. If more than two covariates were being matched or the values of the

Table 4.6: Wallclock time (in seconds) required by each approach to execute all the faulty versions of the specified subject program.

Name	# of Tests	CBI (NM/MDM)	ESP (NM/MDM)
cal	162	179 /181	527/531
col	156	185/186	574/575
comm	186	260/262	799/803
look	193	243/244	722/725
spline	700	500/513	1,557/1,583
tr	870	441/448	1,379/1,391
uniq	431	204/207	646/651
print-tokens	4,130	1,390/1,514	4,757/4,999
print-tokens2	4,115	1,030/1,106	3,252/3,422
replace	395	1,472/1,485	4,650/4,677
schedule	2,710	665/707	1,947/2,039
scheule2	2,650	565/610	1,932/2,030
tcas	1,608	173/183	831/861
totinfo	1,052	187/208	528/568
sed	363	16,136/16,242	53,652/53,859
space	157	5,122/5,227	14,841/15,041
bc	4,000	773/4,231	2,384/9,203
gzip	217	14,225/14,314	46,833/46,994
flex	567	20,640/ 20,931	70,170/70,764
grep	809	13,872/14,285	8,172/8,336
bates	298	694/813	2,312/2,545
heston	316	1,126/1,201	3,753/3,881
mc euro	242	825/837	2,547/2,568
um-olsr	176	618/765	1,829/2,056
ns2	293	1,663/1,839	5,340/5,661
g/g/1	3,000	474/818	1,514/2,311
m/m/c	3,000	386/924	1,178/2,237
mmpp/d/1	3,000	581/1,186	2,111/3,213

covariates were not restricted to '0' or '1' then *MD matching* could not be implemented as a less restrictive form of *exact matching* and the MDM versions of CBI and ESP would be significantly less efficient due to the computation of the $d_M(a, b)$ metric. However, despite an efficient matching implementation, the MDM versions (and EM versions) of CBI and ESP can still result in significant *Slowdown* compared to the versions of ESP and CBI which do not employ matching to balance test cases before estimating the suspiciousness of a predicate. As Figure 4.12 shows the most significant *Slowdown* occurs for the subject programs **bates**, **um-olsr**, **g/g/1**, **m/m/c**, **mmpp/d/1**, **bc** and **totinfo**. All of these subject programs share two characteristics that create significant *Slowdown* when matching is employed: (1) a large number of test cases and (2) a large number of scalar pairs predicates due to many similarly typed variables within the same scope. Combined these characteristics require matching to be performed over a large number of test cases for a large number of predicates for each faulty version of the subject program resulting in almost 7x *Slowdown* for the subject program **bc**.

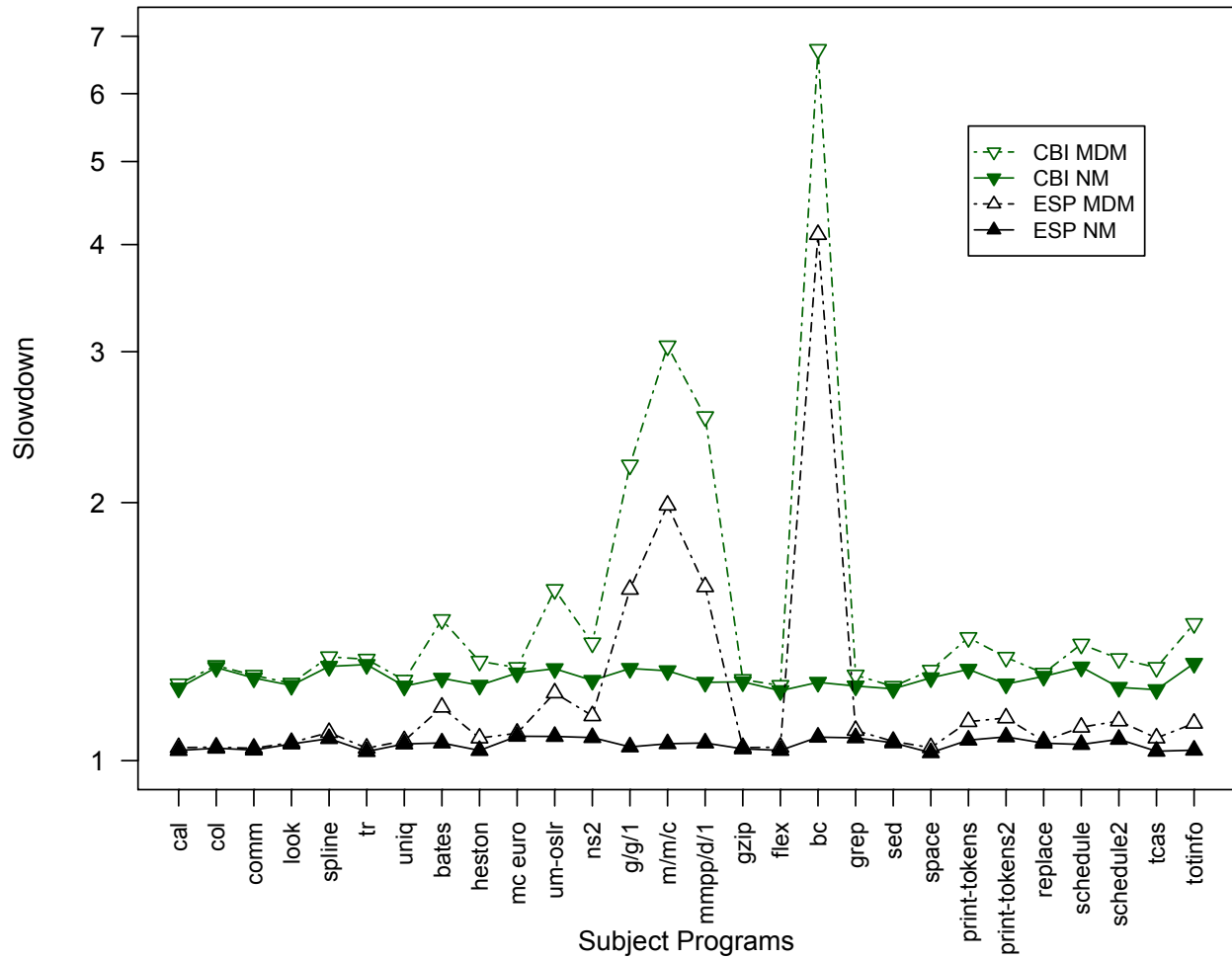


Figure 4.12: *Slowdown* of MDM versions of ESP and CBI compared to NM versions. Note the log scale of the y-axis.

While *exact matching* and *MD matching* can make CBI and ESP less efficient for some subject programs, both approaches require only machine time, not developer time. If the developer can remain productive while matching is performed, overall efficiency will be improved because the developer is given a more effective list of ranked predicates to identify faults. This rationale has been used to argue the effectiveness of formal verification methods despite execution times measured in days and hours [122].

Discussion

Although MDM and EM performed well in the evaluation, each was not as effective as our previous causal model (NM) for a small number of subject program versions. The faults in these versions reflect missing code and are not triggered by the coverage of a statement corresponding to a predicate. This violates an implicit assumption within our approach that the fault must be covered by a predicate to be effectively localized. Recall, NM was less effective than the standard suspiciousness estimates for these same subject program versions in the evaluation in Section 4.4. Similarly, employing *exact matching* and *MD matching* to further reduce confounding bias does not improve the effectiveness of ESP and CBI for programs with missing code faults and in several cases it degrades the ability of the debuggers to localize the fault.

The effectiveness of EM and MDM relative to CBI and ESP is also important to discuss. Again, for the preponderance of the program versions CBI offers more improvement than ESP but for most of the program versions ESP incurs less overall *Cost*. This is because while EM and MDM improve the effectiveness of each statistical debugger, ESP appears to improve less because it is more effective. Similarly, for most of the versions where negative improvement is observed, the effectiveness of ESP degrades less than CBI. Furthermore, given a faulty program version, ESP is more likely than CBI to offer some improvement. This is because the additional predicates employed in ESP create more situations where there is a *lack of balance*. As a result matching has a better chance of improving the effectiveness of ESP than CBI. The validity threats to this portion of the evaluation are very similar to those discussed in Section 4.2.5.

4.5 Chapter Summary

Previous research has shown that two different types of confounding bias (*control flow* and *failure flow*) can detrimentally affect the accuracy of standard suspiciousness estimates employed in statistical debuggers. However, no previous research has reduced both of these biases for standard suspiciousness estimates at the predicate-level. Here, an established approach is taken to predicate-level statistical debugging that enables us to account for factors which create both confounding biases in one causal model. This causal model yields a suspiciousness estimate which can be integrated into each of the standard suspiciousness estimates to significantly improve the fault localization capabilities of ESP and CBI.

Unfortunately, for some predicates the presence of the factors which create confounding bias is very different in those test cases where the predicate is true versus those test cases where the predicate is not true. The differences of the factors in these groups of test cases reflects a *lack of balance* and makes the suspiciousness estimate from our causal model less effective. Two different *statistical matching* approaches

are explored to address this *lack of balance* and improve the effectiveness of the suspiciousness estimates from our causal model: *exact matching* and *Mahalanobis Distance (MD) matching*. Experimental evaluation shows that employing *MD matching* to balance test cases is the most effective approach to reducing control flow and failure flow confounding bias for the suspiciousness estimate of a given predicate. However, for some subject programs with a large number of test cases, employing *MD matching* can result in significant *Slowdown* compared to our original causal model. Despite this inefficiency, if users can remain productive while matching is performed, overall efficiency will be improved since the resulting debugger is significantly more effective.

Chapter 5

Fuzzy Passing Extents

The output of software that employs stochastics can include some natural random variance. Frequently these pieces of software are exploratory simulations, where the precise output for a given test case is not exactly known. Even if test cases with analytical solutions are employed, the output of some simulations, such as Monte Carlo simulations, include some random variance. In these cases, it is difficult to determine if the software passes or fails a given test case. In existing statistical debugging approaches this is required, the labeling of test cases as passing or failing must be a Boolean function. An example helps elucidate this problem and the utility of our solution: *fuzzy passing extents*. Consider the pseudocode implementing the well known Ising model shown in Algorithm 3. The Ising model is used to study ferromagnetism in statistical mechanics. The model consists of discrete variables called spins that can be in one of two states. The spins are arranged in a graph, and each spin interacts with its nearest neighbors [98, 123, 124].

The Monte Carlo implementation of the Ising model shown in Algorithm 3 represents the spin lattice as a 2-dimensional array of integers, each valued at positive or negative one. The program makes randomly proposed changes to these spins, using a Metropolis algorithm to enforce the Boltzmann distribution [98, 123, 124]. The output of the simulation is the final value of *magnetization* – the absolute value of the lattice’s total spin per unit volume – averaged over configurations generated by the Metropolis algorithm. For each input the specified value for the final magnetization is the analytical solution, given by [98, 123, 124].

We generated 100 random, valid inputs to the Ising model. Despite many test cases approaching their analytical solution none of the 100 different test cases exactly matched it. As a result each of the 100 test cases was designated as failing. This creates at least two difficult questions for users tasked with localizing sources of failures in similar situations: (1) does an output have to match the analytical solution exactly to be considered a passing test case? and (2) if not, how close must an output be to the analytical solution to

Algorithm 3 An implementation of the 2-dimensional Ising model.

```

ISING( $w, h, t, N, \beta$ )
1   $L \leftarrow$  (2-D array of  $w \times h$  random spins)
2   $i \leftarrow 0$ 
3   $energy \leftarrow 0$ 
4   $magnetization \leftarrow 0$ 
5  while  $i < N$ 
6      do
7          for (each element in  $L$ )    (a sweep)
8              do
9                   $x \leftarrow$  (random integer  $\in \{0, \dots, w-1\}$ )
10                  $y \leftarrow$  (random integer  $\in \{0, \dots, h-1\}$ )
11                  $\Delta E \leftarrow$  (change in energy from flipping  $L_{x,y}$ )
12                  $r \leftarrow$  (random real  $\in [0, 1]$ )
13                 if ( $r < e^{-\beta \Delta E}$ ), flip  $L_{x,y}$ 
14                 if  $i \geq t$     (the first  $t$  sweeps are for thermalization)
15                     do
16                         for ( $s \in L$ ),  $energy \leftarrow energy +$  (energy of  $s$ )
17                          $currentMagnetization \leftarrow \sum_{j,k} L_{j,k}$ 
18                          $magnetization \leftarrow magnetization + |currentMagnetization|$ 
19                  $i \leftarrow i + 1$ 
20   $energy \leftarrow energy / ((N - t)w^2h^2)$ 
21   $magnetization \leftarrow magnetization / ((N - t)w^2h^2)$ 

```

be considered passing?

In this chapter an approach inspired by fuzzy logic is presented that enables a subject program to both pass and fail a given test case. This approach, *fuzzy passing extents*, enables users to address the two previous questions for software with variance in the output. Thus, by employing *fuzzy passing extents* predicate-level statistical debugging approaches can become effective for software, including many simulations, where it had been previously inapplicable. Section 5.1 gives an overview of fuzzy logic which inspired our *fuzzy passing extents*. Then Section 5.2 presents *fuzzy passing extents*.

5.1 Fuzzy Logic Background

Fuzzy logic is a form of many-valued logic centered around approximate reasoning, as opposed to the exact reasoning featured in traditional logic theory. In traditional logic theory a variable can have one of two truth values: true or false. However, in fuzzy logic variables can have any continuous truth value in the range 0 and 1 ($[0, 1]$). This enables fuzzy logic to process incomplete data and provide approximate solutions to problems traditional logic finds difficult to solve [125, 126, 127].

Basic applications of fuzzy logic characterize subranges of a continuous variable. For example, human age

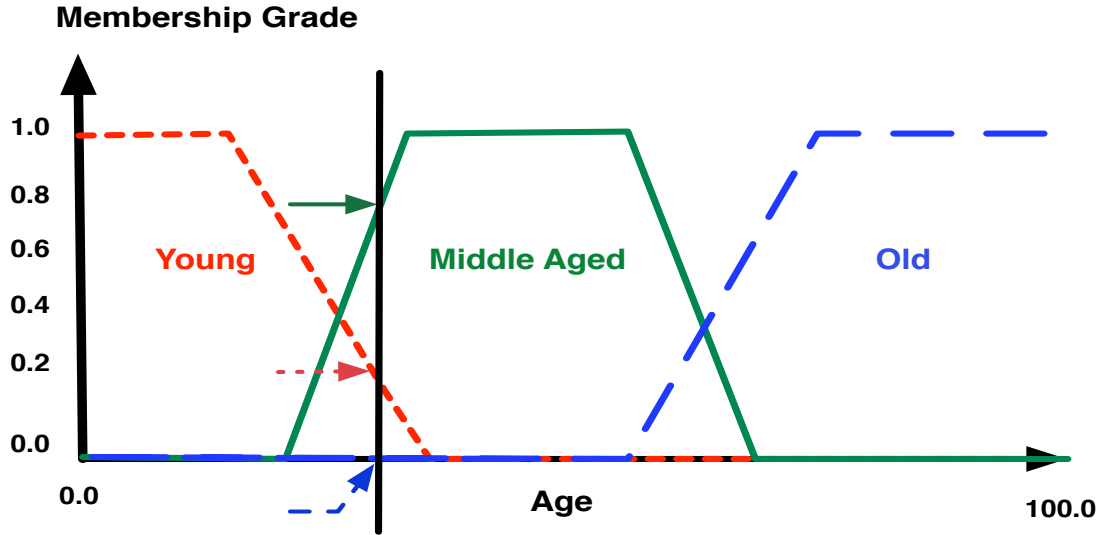


Figure 5.1: Graph of the age scale for the descriptions *young*, *middle-aged*, *old*.

can be described with several separate membership functions defining particular age ranges in which one might qualitatively be considered *young*, *middle-aged* and *old*. Each function maps an age measurement to a *truth value* in the 0 to 1 range ($[0,1]$). These *truth values* are used to determine how one's age might be described.

In Figure 5.1, the meanings of the expressions *young*, *middle-aged*, and *old* are age scale mapping functions. A point on the age scale has three *truth values* — one for each of the three functions (*young*, *middle-aged* and *old*). The vertical line shown in Figure 5.1 represents a particular age and the three arrows reflect the value of each function for the specified age. The blue (*old*) arrow has a truth-value of zero, signifying that the age (35) is *not old*. The red (*young*) arrow has a truth-value of 0.2 signifying that age 35 is *slightly young*. Finally, the green arrow has a truth-value of 0.8 and signifies that age 35 is *fairly middle-aged*.

5.2 Fuzzy Passing Extents

The output of software employing stochastics often includes some random variance. In these cases the precise (or passing) output that is specified for an input is not known. As a result, it is difficult and potentially arbitrary to construct a Boolean function to label the output of test cases as either strictly passing or failing for these pieces of software. Existing predicate-level statistical debuggers require a Boolean function to determine passing output from failing output. This makes these debuggers inapplicable to software with variance in the output.

In this section, an approach inspired by fuzzy logic that enables a subject program to both pass and fail a given test case is presented. The approach, *fuzzy passing extents* remove the requirement that the user must construct a Boolean function to label the outputs of test cases as either passing or failing. *Fuzzy passing extents* allow a user to specify a continuous function instead. *Fuzzy passing extents* assume that the output of a program is (or can easily be transformed to) some ordered set of real numbers. This is typical of most software that employs stochastics, such as exploratory simulations. This set is referred to as the *output list*, X . The passing extent, u , of the output list, X , is computed using Equation 5.1.

$$u = \left(\frac{\sum_{i=1}^{|X|} W_i f_i(X_i)}{\sum_{w \in W} w} \right) \text{ where } f_i : \mathbb{R} \rightarrow [0, 1] \text{ and } |W| = |X|. \quad (5.1)$$

In Equation 5.1, W is an ordered set of weights which focuses attention on particular parts of the *output list*. The functions f_i encode information about the specified (or passing) output and the tolerance for deviation from that output. The passing extent u reflects the extent to which executing a single test case for a faulty subject program produces the specified (or passing) output. The sum of the passing extent of every test case in the test suite reflects the total number of passing test cases. This sum and similar measures derived from u are used in place of the existing terms in the suspiciousness estimates presented in chapters 3 and 4.

Fuzzy passing extents are a continuous generalization of Boolean pass/fail detection [9, 10, 11]. This means that while they do not guarantee improvements in the effectiveness of predicate-level statistical debuggers, *fuzzy passing extents* can reproduce the results of each of the suspiciousness estimates previously presented in this dissertation. This backwards compatibility is described next.

Consider a program where the output for each test case is mapped to a single real number X , and the passing output for the test case is the real number X_{pass} . The delta function function, $f(x) = \delta_{X_{pass}}(X)$, will separate test cases that pass ($u = 1$) from test cases that fail ($u = 0$). The choice of $f(x)$ is important, $\delta_{X_{pass}}(X) = 1$ if X exactly matches X_{pass} , otherwise $\delta_{X_{pass}}(X) = 0$.

$$u_{TC_j} = \left(\frac{\sum_{i=1}^{|X|} W_{i,TC_j} f_{i,TC_j}(X_{i,TC_j})}{\sum_{w \in W} w} \right) \text{ where } f_{i,TC_j} : \mathbb{R} \rightarrow [0, 1] \text{ and } |W| = |X|. \quad (5.2)$$

This distinction between passing and failing test cases is the same as the distinction made in the terms used in the suspiciousness estimates in chapters 3 and 4. This backwards compatibility between the *fuzzy*

passing extent and the terms used in the previous presented estimates (s , f , s_p , f_p , $s_{p\text{ obs}}$ and $f_{p\text{ obs}}$) is shown in equations 5.3 - 5.8, where TC is the test suite and u_{TC_j} , shown in Equation 5.2, is the *fuzzy passing extent* of test case j . In Equation 5.2, W_{TC_j} is an ordered set of weights for different parts of the output list for test case TC_j . The functions f_{i,TC_j} encode information about the specified (or passing) output for test case TC_j and the tolerance for deviation from that output.

$$s = \sum_{j=1}^{|TC|} u_{TC_j} \quad (5.3)$$

$$f = \sum_{j=1}^{|TC|} 1 - u_{TC_j} \quad (5.4)$$

$$s_p = \sum_{j=1}^{|TC|} \begin{cases} u_{TC_j}, & \text{if } p \text{ is true in test case } TC_j \\ 0, & \text{if } p \text{ is not true in test case } TC_j \end{cases} \quad (5.5)$$

$$f_p = \sum_{j=1}^{|TC|} \begin{cases} 1 - u_{TC_j}, & \text{if } p \text{ is true in test case } TC_j \\ 0, & \text{if } p \text{ is not true in test case } TC_j \end{cases} \quad (5.6)$$

$$s_{p\text{ obs}} = \sum_{j=1}^{|TC|} \begin{cases} u_{TC_j}, & \text{if } p \text{ is evaluated (true or false) in test case } TC_j \\ 0, & \text{if } p \text{ is not evaluated (true or false) in test case } TC_j \end{cases} \quad (5.7)$$

$$f_{p\text{ obs}} = \sum_{j=1}^{|TC|} \begin{cases} 1 - u_{TC_j}, & \text{if } p \text{ is evaluated (true or false) in test case } TC_j \\ 0, & \text{if } p \text{ is not evaluated (true or false) in test case } TC_j \end{cases} \quad (5.8)$$

It is important to note that the measures in equations 5.3 - 5.8 are not directly applicable to the causal models presented in Chapter 4 which estimate the suspiciousness of instrumented predicates. Recall, three types of measures are included in these models: *outcome* variables, *treatment* variables and *covariate* terms. Each of these measures was described as a Boolean (0/1) measure when presented in Chapter 4. When *fuzzy passing extents* are employed in statistical debuggers using causal models only the *outcome* variable becomes continuous; the *treatment* variable and *covariates* remain Boolean measures. This is because the outcome variable is the only term in the causal model that measures the extent to which a test case for a faulty subject program passes. The treatment variable and covariates still reflect Boolean measures.

Ultimately, choosing the functions for the passing extent for the test cases of a faulty subject program is a nontrivial problem, and one that is entirely dependent on the user. However, the evaluation in Section 5.4

shows the fuzzy formalism *allows* a choice that can outperform the Boolean function used to label passing and failing test cases in existing statistical debugging approaches. Furthermore, while *fuzzy passing extents* do not guarantee improvements, equations 5.3 - 5.8 show that they can reproduce the results of each of the existing suspiciousness estimates presented in chapters 3 and 4 for software which employ a Boolean function to label passing and failing test cases. This ensures that debuggers employing *fuzzy passing extents* reasonably will not be outperformed by their traditional counterparts which do not employ *fuzzy passing extents*.

5.3 Fuzzy Passing Extent Example

An example helps elucidate the utility of *fuzzy passing extents*. Recall, the implementation of the 2-dimensional Ising model presented in the chapter introduction. The implementation represents the spin lattice as a 2-dimensional array of integers, each valued at positive or negative one. Random changes are proposed to these spins using a Metropolis algorithm to enforce the Boltzmann distribution [98, 123, 124]. The output of the simulation is the final value of *magnetization* – the absolute value of the lattice’s total spin per unit volume – averaged over configurations generated by the Metropolis algorithm.

As previously discussed, this implementation of the Ising model should approximate the analytical solution, given by [98, 123, 124]. However, the following fault, based on published mistakes, is injected into the simulation: the absolute value of each sweep’s magnetization is not taken while summing magnetizations to compute the average (line 18 of the pseudo-code) [128]. Over many sweeps, one would expect the faulty behavior to always result in zero magnetization because the algorithm would thoroughly sample states magnetized in positive and negative directions. However, over fewer sweeps, the magnetization will often be close to plus or minus the analytical solution. The reason for this is that both the positive and negative magnetizations are surrounded by relatively probable configurations. If the random walk of the Metropolis algorithm wanders toward one direction of magnetization, it has a low probability of moving to the other within a small number of moves. The result is an implementation of the Ising model that produces output that approaches the analytical solution at times and output that significantly veers away from the analytical solution at others. This faulty implementation of the Ising model is shown in Algorithm 4.

We apply a version of ESP employing *fuzzy passing extents* to the faulty implementation of the 2-dimensional Ising model shown in Algorithm 4. The suspiciousness of the predicates instrumented by ESP are estimated using the causal model which controls for the control and failure flow confounding biases presented in Chapter 4. The outcome variable for the model is determined by the fuzzy passing function f . The passing function f is a bell curve centered on the analytical solution, \bar{x} , for inputs specified in each of the

100 test cases we generated in the introduction. Formally, f is $\exp(-(x - \bar{x})^2/a^2)$ where a is the tolerance for deviation. For this example $a = 1.0$.

Algorithm 4 A faulty implementation of the 2-dimensional Ising model.

```

ISING( $w, h, t, N, \beta$ )
1   $L \leftarrow$  (2-D array of  $w \times h$  random spins)
2   $i \leftarrow 0$ 
3   $energy \leftarrow 0$ 
4   $magnetization \leftarrow 0$ 
5  while  $i < N$ 
6      do
7          for (each element in  $L$ )    (a sweep)
8              do
9                   $x \leftarrow$  (random integer  $\in \{0, \dots, w - 1\}$ )
10                  $y \leftarrow$  (random integer  $\in \{0, \dots, h - 1\}$ )
11                  $\Delta E \leftarrow$  (change in energy from flipping  $L_{x,y}$ )
12                  $r \leftarrow$  (random real  $\in [0, 1]$ )
13                 if ( $r < e^{-\beta \Delta E}$ ), flip  $L_{x,y}$ 
14                 if  $i \geq t$     (the first  $t$  sweeps are for thermalization)
15                     do
16                         for ( $s \in L$ ),  $energy \leftarrow energy +$  (energy of  $s$ )
17                          $currentMagnetization \leftarrow \sum_{j,k} L_{j,k}$ 
18                          $magnetization \leftarrow magnetization + currentMagnetization$  // faulty line
19                  $i \leftarrow i + 1$ 
20   $energy \leftarrow energy / ((N - t)w^2h^2)$ 
21   $magnetization \leftarrow magnetization / ((N - t)w^2h^2)$ 

```

The suspiciousness estimates for the static predicate ($magnetization < 0$)₁₈ in ESP localize the values of *magnetization* (in line 18 of the pseudo-code) that are < 0 . The passing extents for this predicate are significantly lower than the other predicates because repeated negative values of *magnetization* cause the simulation to significantly deviate from the analytical solution. This analysis is not possible without *fuzzy passing extents* because none of the 100 test cases exactly match the analytical solution for the Ising model. As a result the suspiciousness estimates for each of the instrumented predicates in the traditional version of ESP are the same. Recall, this was also true for the correct implementation of the Ising Model presented in the introduction. This example illustrates how important it is to make statistical debugging approaches applicable to programs with variance in their output. In Section 5.4 the importance of *fuzzy passing extents* is further supported with a more complete evaluation.

5.4 Evaluation

This section evaluates employing *fuzzy passing extents* in the statistical debuggers Tarantula, ESP and CBI. Sections 5.4.1 and 5.4.2 describe the subject programs and statistical debuggers included in the evaluation. Then, sections 5.4.3 and 5.4.4 evaluate the effectiveness and efficiency of employing *fuzzy passing extents* in the debuggers. Finally, Section 5.4.5 discusses the validity of the evaluation.

5.4.1 Subject Programs

Recall from previous chapters that the Siemens suite, a set of fault localization benchmark composed of eight programs with 129 faulty versions, can be employed to evaluate statistical debuggers [7, 12, 22, 23, 31, 32]. In this evaluation a subset of the Siemens suite consisting of three programs (`tcas`, `schedule` and `schedule2`) and 59 faulty versions is employed to evaluate the efficiency and effectiveness of employing *fuzzy passing extents* in established statistical debuggers.

In this evaluation the programs `tcas`, `schedule` and `schedule2` are adapted to include stochastics. For the `tcas` program the values of constants are sampled from a uniform distribution where the minimum value is half of the value of the constant and the maximum value is one and a half the value of the constant. In the two priority queue scheduler programs (`schedule` and `schedule2`) the programs are modified to include arrival and service times drawn from a normal distribution for each process. The unmodified test cases for these programs dictate the order in which processes are scheduled in the queue. These modifications represent the uncertainty of measurements often used in exploratory simulations that create variance in program output. However, each faulty version of each program contains the same faults as the original faulty program version in the Siemens suite [118]. These programs were chosen because straightforward adaptations could be made to create subject programs that featured: (1) some variance in the output and (2) faults from an established set of benchmarks.

The Traffic Collision Avoidance System, `tcas`, monitors an aircraft's airspace and warns pilots of possible collisions via three different outputs: (0) adjust the trajectory of the aircraft downwards, (1) do not adjust the trajectory of the aircraft or (2) adjust the trajectory of the aircraft upwards. For the traditional debuggers, a test case for a faulty version of `tcas` passes if it directly matches the output that was specified for the deterministic version of the `tcas` program. For the statistical debuggers employing *fuzzy passing extents* the passing function, f , shown in Table 5.1 is employed. The function shown in Table 5.1 does not consider a test case to fully fail unless it drastically differs from the specified output. This occurs when a faulty version instructs the pilot to: (a) adjust the trajectory of the aircraft downwards, when an upwards adjustment is specified or (b) adjust the trajectory of the aircraft upwards, when a downwards adjustment is specified.

Table 5.1: The passing function used for `tcas` test cases.

Expected Output	Faulty Version Output	Passing Extent (u)
0	0	1.0
0	1	0.5
0	2	0.0
1	0	0.5
1	1	1.0
1	2	0.5
2	0	0.0
2	1	0.5
2	2	1.0

Table 5.2: Subject programs used in the evaluation of fuzzy passing extents

Name	# of Vers Used / # of Vers	LoC	# of Tests	Description
<code>schedule</code>	9/9	292	2,710	priority scheduler
<code>scheule2</code>	9/10	301	2,650	priority scheduler
<code>tcas</code>	41/41	141	1,608	altitude separator

The output of the two scheduling benchmarks (`schedule` and `schedule2`) is a list of the identification numbers of processes in the order they exit the queue. For the traditional debuggers a test case passes if the ordered output of the faulty version exactly matches the ordered output specified for the test case. If this is not true, then the test case fails. However, for the statistical debuggers employing *fuzzy passing extents* the Levenshtein distance between the ordered output of the faulty version and the ordered output specified for the test case serves as the passing function. The Levenshtein distance between the output and the specified output is the minimum number of edits needed to transform one list of process identification numbers into the other, where the edit operations are insertion, deletion, or substitution of process identification numbers [129]. The passing function, which computes the Levenshtein distance between the faulty version output (out) and the specified output (exp) is shown in Algorithm 5.

Recall, all of the faults included in these subject programs are computation related, as opposed to memory-related. These faults reflect operator and operand mutations, missing and extraneous code, and constant value mutations. For subject program versions with a faulty constant assignment statement, the assignment statement is considered to be examined by a developer when it is directly examined or when a statement explicitly using the constant is examined. For subject program versions where the fault reflects a missing statement, statements directly adjacent to the missing code qualify as the missing statement. These issues are handled the same way they are in the published work of other researchers in the fault localization community [7, 12, 22, 23].

Algorithm 5 The passing function used for `sched` and `sched2` test cases to compute Levenshtein Distance.

LEVENSHTEINDISTANCE(*out*, *exp*)

```

1  d is an empty  $|out| + 1$  by  $|exp| + 1$  matrix
2  // di,j will be the distance between (out0 ... outi) and (exp0 ... expj)
3  for (i from 0 to  $|out|$ )
4      do
5          di,0 ← i
6
7  for (i from 0 to  $|exp|$ )
8      do
9          d0,j ← i
10
11 for (j from 1 to  $|exp|$ )
12     do
13         for (i from 1 to  $|out|$ )
14             do
15                 if outi = expj
16                     then
17                         di,j ← di-1,j-1 // no operation required, distance does not increase
18                     else
19                         delete ← di-1,j + 1
20                         insert ← di,j-1 + 1
21                         subs ← di-1,j-1 + 1
22                         di,j ← min(delete, insert, subs)
23  maxSize ← max( $|out|$ ,  $|exp|$ )
24  return  $1 - (\frac{d_{|out|,|exp|}}{maxSize})$ 
```

5.4.2 Competing Statistical Debuggers

Three fault localization approaches are included in the evaluation: Cooperative Bug Isolation (CBI), Exploratory Software Predictor (ESP), and Tarantula. While these approaches are presented in Chapter 3, they are reviewed here and a version of each which employs *fuzzy passing extents* is presented.

Fuzzy Cooperative Bug Isolation (CBI)

In the evaluation, the fuzzy and traditional versions of CBI are employed with the static *single variable*, static *scalar pairs* and *branches* instrumentation schemes described in sections 3.3.1, 3.3.2 and 3.4.2. Suspiciousness is estimated in both versions of CBI using the causal model presented in Chapter 4 which controls for control flow and failure flow confounding bias. Recall, in the traditional version of CBI, the treatment variable in the causal model, Y , is set to 0 if a test case passes and 1 if a test case fails. However, here in the version of CBI employing *fuzzy passing extents*, the treatment variable for a test case Y is set to $1 - u$, where u is the passing extent determined by the passing function for the subject program. Matching of test cases for the causal model is performed just as it is presented in Chapter 4.

Given a list of predicates ranked by suspiciousness estimate, the statements in the traditional version of CBI and the version employing *fuzzy passing extents* are ranked according to the following:

1. For each statement, $stmt$, identify the corresponding predicate with the highest suspiciousness estimate, $stmt_{high}$.
2. Move the statement, $stmt$ and highest suspiciousness estimate, $stmt_{high}$, to set ST .
3. Rank the statements ST in descending order by suspiciousness estimate.

Fuzzy Exploratory Software Predictor (ESP)

The predicates employed in the traditional version of ESP and the version of ESP employing *fuzzy passing extents* are a superset of the predicates employed in CBI. Along with the static *single variable*, static *scalar pairs* and *branches* instrumentation schemes employed in CBI, ESP also employs the reduced FAST elastic *single variable* and *scalar pairs* instrumentation schemes described in 3.4.9. The suspiciousness of the predicates and the ranking of the program statements in ESP is computed in the same manner as it is in CBI.

Fuzzy Tarantula

Recall, in contrast to the predicative-level statistical debuggers CBI and ESP, Tarantula is a statement-level debugger. For a given statement, $stmt$, and a subject program where TC is the test suite and u_{TC_j} is the *fuzzy*

Table 5.3: Measures used by the traditional and fuzzy versions of Tarantula

Name	Non-Fuzzy Description	Fuzzy Description
s_{stmt}	# of passing test cases that execute $stmt$	$\sum_{j=1}^{ TC } \begin{cases} u_{TC_j}, & \text{if } stmt \text{ is true in test case } TC_j \\ 0, & \text{if } stmt \text{ is not true in test case } TC_j \end{cases}$
f_{stmt}	# of failing test cases that execute $stmt$	$\sum_{j=1}^{ TC } \begin{cases} 1 - u_{TC_j}, & \text{if } stmt \text{ is true in test case } TC_j \\ 0, & \text{if } stmt \text{ is not true in test case } TC_j \end{cases}$
s	# of successful test cases	Equation 5.3
f	# of failing test cases	Equation 5.4

passing extent of test case j , Table 5.3 shows the measures the traditional version of Tarantula and version of Tarantula employing *fuzzy passing extents* use to estimate the suspiciousness of $stmt$. The computation which aggregates these measures into a suspiciousness estimate is shown in Equation 3.9. Once the suspiciousness of each statement that is executed in the subject test suite is estimated, all of the executed statements are ranked in descending order of suspiciousness [7].

5.4.3 Effectiveness

To study the effectiveness of employing *fuzzy passing extents* in the statistical debugging approaches described in Section 5.4.2, a version of the established metric *Cost* is employed [7, 8, 22, 23, 28, 31, 32]. Given the ranked set of statements, *Cost* measures the percentage of executed statements a developer must examine before encountering the faulty statement. Recall, if there are ties, it is assumed that the developer must examine all of the tied statements. For example, if there are n executed statements in a program and all n statements have the same suspiciousness estimate, it is assumed that the developer must examine all n statements. A lower score is preferable because it means that fewer of the statements must be considered before the faulty statement is found. Due to the variance in the output of the subject programs used in the evaluation, a version of *Cost* which reflects the average *Cost* of localizing the fault in a version of the subject program over 100 different trials is used. This cost metric is referred to as \overline{Cost} .

The effectiveness of employing *fuzzy passing extents* in the statistical debuggers included in the evaluation for the subject programs is shown in Table 5.4. Each row in Table 5.4 shows a \overline{Cost} range, and the number (and percentage) of subject programs for each approach that incur the specified \overline{Cost} . Figure 5.2 provides a graphical view of this data. In Figure 5.2 the x-axis represents the lower bound of each \overline{Cost} range and the y-axis represents the percentage of subject programs where a \overline{Cost} less than or equal to the upper bound is incurred. This presentation of data follows the established convention in the fault localization community [7, 12, 22, 23].

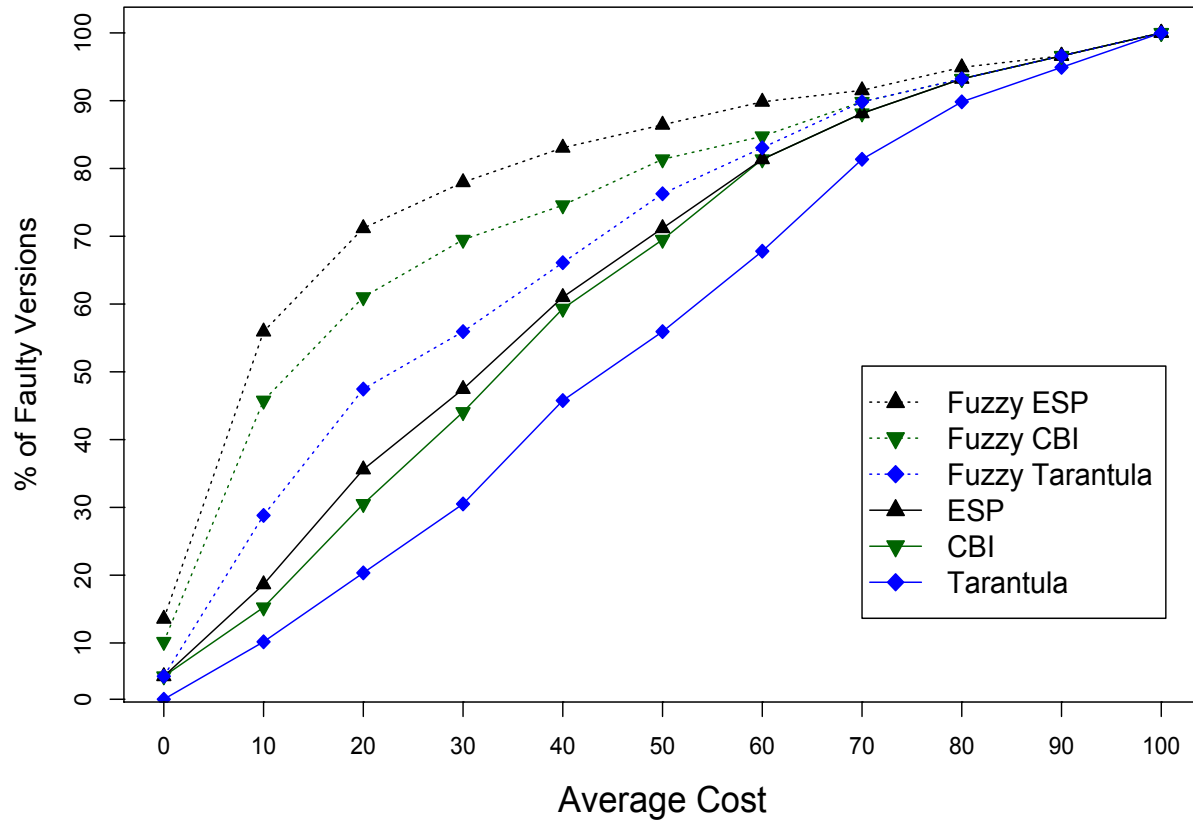


Figure 5.2: Evaluation of the effectiveness of *fuzzy passing extents* in Tarantula, CBI and ESP.

Table 5.4: Number (percentage) of faulty version ranked statement lists in each score range for all approaches.

Cost	Tarantula # (%) Progs	F. Tarantula # (%) Progs	CBI # (%) Progs	F. CBI # (%) Progs	ESP # (%) Progs	F. ESP # (%) Progs
0 - 1%	1 (1.69%)	3 (5.08%)	3 (5.08%)	6 (11.28%)	3 (5.08%)	8 (13.56%)
1 - 10%	5 (8.47%)	14 (23.73%)	6 (10.17%)	21 (35.59%)	8 (13.56%)	25 (42.37%)
10 - 20%	6 (10.17%)	11 (13.41%)	9 (15.25%)	9 (14.02%)	10 (5.90%)	9 (15.25%)
20 - 30%	6 (10.17%)	5 (8.47%)	8 (13.56%)	5 (8.47%)	7 (11.86%)	4 (6.78%)
30 - 40%	9 (15.25%)	6 (10.17%)	9 (15.25%)	3 (5.08%)	8 (13.56%)	3 (5.08%)
40 - 50%	6 (10.17%)	6 (10.17%)	6 (10.17%)	4 (6.78%)	8 (13.56%)	2 (3.39%)
50 - 60%	7 (11.86%)	4 (6.78%)	7 (11.86%)	2 (3.39%)	6 (10.17%)	2 (3.39%)
60 - 70%	8 (13.56%)	4 (6.78%)	8 (13.56%)	3 (13.56%)	4 (6.78%)	1 (1.69%)
70 - 80%	5 (8.47%)	2 (3.39%)	3 (5.08%)	2 (3.39%)	3 (5.08%)	2 (3.39%)
80 - 90%	3 (5.08%)	2 (3.39%)	2 (3.39%)	2 (3.39%)	2 (3.39%)	1 (1.69%)
90 - 100%	3 (5.08%)	2 (3.39%)	2 (3.39%)	2 (3.39%)	2 (3.39%)	2 (3.39%)

Fuzzy CBI

Fuzzy CBI performs better than CBI for the three subject programs included in the evaluation. There are only 2 out of the total 59 faulty program versions where Fuzzy CBI assigned a lower rank to the statement

containing the fault than CBI. In these versions the *fuzzy passing extents* used to calculate the suspiciousness estimates do not localize the faulty statement as well as their traditional counterparts. These are outlying data points in the evaluation (<4% of all faulty versions). For the vast majority of the faulty versions, the *fuzzy passing extents* used to compute the outcome variables in the causal models in Fuzzy CBI leads to more effective fault localization in the adapted subset of the Siemens suite. Furthermore, for the 2 versions where CBI outperforms Fuzzy CBI the fault in the version reflects missing code. Recall from Section 4.2.5, that other attempts to improve the effectiveness of statistical debuggers have failed to localize these types of faults well because the faults violate the coverage trigger assumption [32, 31]. This is also the case for *fuzzy passing extents*.

Overall the traditional version of CBI struggles in this portion of the evaluation because it classifies significantly more test cases as failing than Fuzzy CBI. As a result, most of the CBI instrumented predicates are assigned the same or very similar suspiciousness estimates. These similarities and ties in suspiciousness result in poor \overline{Cost} scores because very few predicates are separated from the masses.

Fuzzy ESP

The results of the evaluation for Fuzzy ESP are similar to the results of the evaluation Fuzzy CBI: Fuzzy ESP is more effective than its traditional counterpart, ESP. Also, for the same 2 faulty program versions where Fuzzy CBI is outperformed by traditional CBI, Fuzzy ESP is outperformed by traditional ESP. Furthermore, most of the elastic predicates in the traditional version of ESP are assigned the same or similar suspiciousness estimates, just as the static predicates in CBI were. Again, this is because the traditional version of ESP classifies significantly more test cases as failing than Fuzzy ESP. As a result, the elastic predicates in traditional ESP are ineffective and there is very little difference between the effectiveness of the traditional version of ESP and the traditional version of CBI. In contrast, the *fuzzy passing extents* employed in Fuzzy ESP do not render the elastic predicates ineffective. Instead, the improvements in effectiveness seen for Fuzzy ESP compared to Fuzzy CBI in the evaluation are similar to the improvements observed for the traditional version of ESP compared to the traditional version of CBI for subject programs without variance in the output.

Fuzzy Tarantula

Fuzzy Tarantula also outperforms its traditional counterpart in the evaluation. Recall, both versions of Tarantula only profile statement coverage data as opposed to the traditional and *fuzzy passing extent* versions of ESP and CBI which employ predicates to profile variable values. As a result the traditional version of Tarantula is less effective than the traditional version of ESP or CBI. Also, the fuzzy version of Tarantula is less effective than the fuzzy version of ESP or CBI.

Table 5.5: Average wallclock time (in seconds) required by each approach to execute all the faulty versions of the specified subject program over 100 trials.

Name	CBI (Traditional/Fuzzy)	ESP (Traditional/Fuzzy)	Tarantula (Traditional/Fuzzy)
<code>schedule</code>	707/757	2,039/2,088	30/81
<code>scheule2</code>	610/ 658	2,030/2,077	30/79
<code>tcas</code>	183/186	861/864	12/15

It is important to note the particularly poor performance of the traditional Tarantula debugger in this portion of the evaluation. The performance of Tarantula is worse than the expected performance of a simple-minded approach to fault localization which takes all statements included in failing test cases and ranks them in a random order. Such an approach would be expected to localize faults in the same percentage of faulty versions as the \overline{Cost} incurred. For example at a \overline{Cost} of searching through 50% of the source code in each of the faulty versions, the simple-minded approach would be expected to localize faults in 50% of the faulty versions. Similarly at a \overline{Cost} of searching through 30% of the source code of the faulty versions it would be expected to localize faults in 30% of the faulty versions.

The traditional version of Tarantula does not outperform the simple-minded approach because there are no ties in terms of suspiciousness estimates in the ranked list of statements in the simple-minded approach. In the traditional version of Tarantula ties occur frequently in ranked list of statement in this portion of the evaluation because many test cases are classified as failing. Since the \overline{Cost} metric requires all tied statements to be examined before the fault is considered to be localized, the effectiveness of Tarantula in this portion of the evaluation is particularly poor.

5.4.4 Efficiency

Here, the efficiency of *fuzzy passing extents* for each of the statistical debuggers included in the evaluation is analyzed. Table 5.5 shows the average wallclock time for ranking the statements for each faulty version of the subject programs included in the evaluation using the traditional and *fuzzy passing extent* versions of Tarantula, and ESP and CBI.

Figure 5.3 shows the data in Table 5.5 graphically. However, instead of measuring efficiency in terms of wallclock time, Figure 5.3 measures efficiency in terms of *Slowdown*. Recall, *Slowdown* is a relative efficiency measure which compares the average wallclock time of an approach for a subject program to the average wallclock time for the same subject program using the traditional version of Tarantula. The formula for *Slowdown* is shown in Equation 3.10. Because the traditional version of Tarantula is the most efficient approach in the evaluation the *Slowdown* of the other approaches is always >1 .

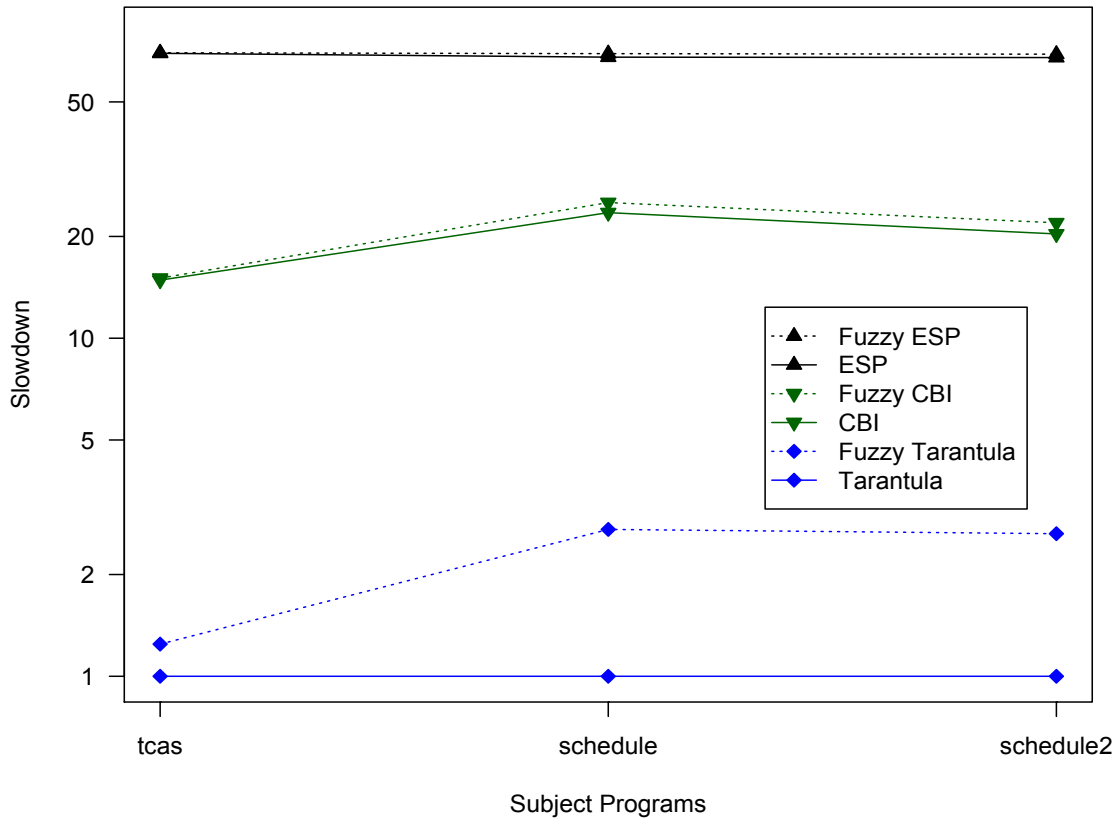


Figure 5.3: Evaluation of the efficiency of *fuzzy passing extents* in Tarantula, CBI and ESP. Note the log scale of the y-axis.

Figure 5.3 reveals several trends in the relative efficiency of the statistical debugging approaches employing *fuzzy passing extents*. The most evident trend is that the *Slowdown* incurred when employing *fuzzy passing extents* is very dependent on the passing function chosen by the user. As Table 5.5 and Figure 5.3 show, the simple passing function used by the debugging approaches for localizing faults in the `tcas` subject program incurs less *Slowdown* than the complex Levenshtein distance passing function for `schedule` and `schedule2`. As a result the *Slowdown* of each version of each approach which employs *fuzzy passing extents* is greater for the subject programs `schedule` and `schedule2` than for the subject program `tcas`.

The increase in *Slowdown* when moving from the simple `tcas` passing function to the more complex Levenshtein distance passing function for `schedule` and `schedule2` is more evident in the version of Tarantula which employs *fuzzy passing extents* than the corresponding versions of CBI and ESP. This is because in ESP and CBI the additional computational overhead incurred for the passing function is masked by the use

of the causal models and matching to estimate the suspiciousness of the predicates. The statement-level implementation of Tarantula employed in our evaluation does not employ these predicate-level techniques. As a result Fuzzy Tarantula does not mask any inefficiencies caused by complex passing functions.

5.4.5 Validity

Internal, external, and construct validity threats affect the evaluation of employing *fuzzy passing extents* in the statistical debuggers CBI, ESP and Tarantula. Threats to internal validity arise when factors affect the dependent variables without the evaluators' knowledge. It is possible that some flaws in the implementation of CBI, ESP and Tarantula could have affected the results of the evaluation. However, the results for these statistical debuggers in other evaluations in this dissertation with different subject programs closely match results from other researchers [7, 13, 22]. Threats to external validity occur when the results of the evaluation cannot be generalized. The evaluation was performed on only 3 subject programs with a total of 59 versions. These programs were chosen because straightforward adaptations could be made to them to create subject programs that featured: (1) variance in the output and (2) faults from an established set of benchmarks. However, the effectiveness of *fuzzy passing extents* for CBI, ESP and Tarantula cannot be generalized to other faults in other subject programs. This is of particular concern for *fuzzy passing extents* because their effectiveness can significantly depend on a user's ability to define an effective passing function. However, while *fuzzy passing extents* do not guarantee improved effectiveness, Section 5.2 shows that it is straightforward to define a passing function that reproduces existing suspiciousness estimates. Threats to construct validity concern the appropriateness of the \overline{Cost} metric used in the evaluation. While the *Cost* metric is well established in the fault localization community and the \overline{Cost} metric is a straightforward adaption of it for subject programs with variance in their output, more human user studies into how useful developers find statement-ranking metrics need to be performed [114]. However, the more accurate statistical debugging methods are, the more meaningful such human user studies will become.

5.5 Chapter Summary

Previous approaches to statistical debugging have not assumed any variance from the specified (or passing) output for a given test case. This assumption precludes the practical consideration of a large class of software, including exploratory simulations, which employ stochastics to reflect uncertainty in an underlying model. The *fuzzy passing extents* presented in this chapter directly address this limitation. They enable users to define a function which specifies the degree ($[0, 1]$) to which a subject program passes a given test case. This capability allows the output of a faulty subject program to both pass and fail for a given test case. This is

fundamentally different from existing statical debuggers which require the user to specify a Boolean function to label the output of a subject program as strictly passing (1) or failing (0) for a given test case.

The application of *fuzzy passing extents* does not guarantee improved effectiveness for a statistical debugger. However, the evaluation in Section 5.4.3 shows that it is possible to define straightforward passing functions for subject programs with varying output. Furthermore, these passing functions enable *fuzzy passing extents* to significantly improve the effectiveness of existing statistical debuggers compared to their traditional counterparts which do not employ *fuzzy passing extents*.

Section 5.4.4 shows that the additional computational overhead incurred by employing *fuzzy passing extents* in a statistical debugger is largely dependent on the user's choice of the passing function. However, we hypothesize that many users will be able to identify efficient passing functions and that when users specify complex passing functions which incur significant overhead, the effectiveness of the debugger will also significantly improve.

Chapter 6

Discussion

The contributions in chapters 3, 4 and 5 enable developers to localize faults causing failures more effectively than existing alternatives. Each of these research contributions offers different improvements and incurs different overheads. Here we dissect and discuss the improvements and overheads of each of the research contributions presented in this dissertation. Both software employing floating-point computations and continuous stochastic distributions and general purpose software are considered. An evaluation which tests if the total time required by users to localize faults in subject programs is reduced when using our research contributions versus existing tools is also presented. That evaluation serves as the overall measure of success for our work.

6.1 Effectiveness

The effectiveness of the statistical debugger Exploratory Software Predictor (ESP), which implements our research contributions, is measured via the established metric *Cost* [7, 8, 22, 23, 28, 31, 32]. Recall, given a ranked set of statements, *Cost* measures the percentage of statements a developer must examine before encountering the faulty statement. If there are ties, it is assumed that the developer must examine all of the tied statements. For example, if there are n executed statements in a program and all n statements have the same suspiciousness estimate, it is assumed that the developer must examine all n statements. Therefore the *Cost* of finding the fault is 100%. A lower score is preferable because it means that fewer of the statements must be considered before the faulty statement is found.

The effectiveness of the statistical debuggers Tarantula, Cooperative Bug Isolation (CBI) and ESP for the 31 subject programs and 387 faulty program versions used in this dissertation, is shown in Table 6.1. The versions of Tarantula and CBI reflect the original published implementations and do not take advantage

Table 6.1: Number (percentage) of faulty version ranked statement lists in each score range for all approaches.

Cost	Tarantula # (%) of Programs	ESP # (%) of Programs	CBI # (%) of Programs
0% - 1%	14 (3.62%)	70 (18.09%)	43 (11.11%)
1% - 10%	89 (23.00%)	163 (42.12%)	139 (35.92%)
10% - 20%	72 (18.60%)	62 (16.02%)	50 (12.92%)
20% - 30%	33 (8.53%)	13 (3.36%)	39 (10.08%)
30% - 40%	54 (13.95%)	29 (7.49%)	18 (4.65%)
40% - 50%	27 (6.98%)	10 (2.58%)	27 (6.98%)
50% - 60%	14 (3.62%)	8 (2.07%)	15 (3.88%)
60% - 70%	21 (5.43%)	10 (2.58%)	17 (4.39%)
70% - 80%	12 (3.10%)	7 (1.81%)	21 (5.43%)
80% - 90%	16 (4.13%)	11 (2.84%)	14 (3.62%)
90% - 100%	35 (9.04%)	4 (1.03%)	4 (1.03%)

of any of our contributions. ESP expands upon a version of CBI and it employs each contribution in this dissertation.

The 31 subject programs used in this evaluation are described in Table 3.4 and Table 5.2. Each row in Table 6.1 shows a *Cost* range, and the number (and percentage) of subject programs for each debugger that incur the specified *Cost*. Figure 6.1 provides a graphical view of this data. In Figure 6.1 the x-axis represents the lower bound of each *Cost* range and the y-axis represents the percentage of subject programs where a *Cost* less than or equal to the upper bound is incurred. This presentation of data follows the established convention in the fault localization community [7, 12, 22, 23].

Table 6.1 and Figure 6.1 show that ESP is significantly more effective at localizing faults than CBI and Tarantula for all the subject programs included in this dissertation. This result is not surprising. ESP takes advantage of the elastic predicates presented in Chapter 3, the reduced biased suspiciousness estimation presented in Chapter 4 and the fuzzy passing extents presented in Chapter 5. Combined these three research contributions enable ESP to localize faults significantly more effectively than the existing alternatives.

Recall, addressing the deficiencies of existing alternative debuggers, such as CBI and Tarantula, for programs employing floating-point computations and continuous stochastic distributions, is the main goal of our work. Further exploration highlights ESP's fault localization capabilities when evaluation is restricted to these types of programs.

The eight widely used but faulty simulations introduced in Section 3.4.4, and the three Siemens Suite benchmarks which were adapted to include variance in their output in Section 5.4.1, reflect programs employing floating-point computations and continuous stochastic distributions. These programs are shown in the eight bottom rows of Table 3.4 and in Table 5.2. In this portion of the evaluation the effectiveness of ESP, CBI and Tarantula is tested for only these eleven subject programs and 67 faulty versions.

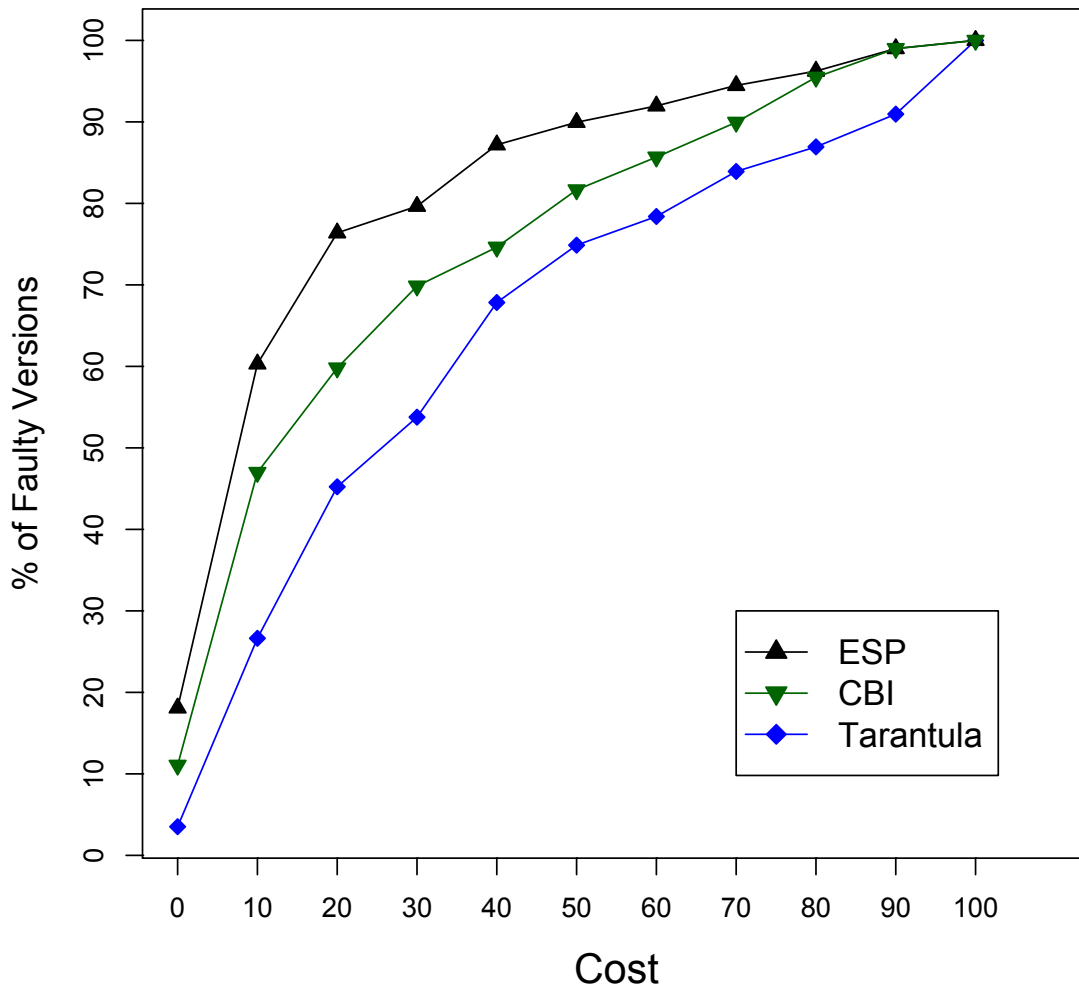


Figure 6.1: Comparison of the effectiveness of Tarantula, CBI and ESP using the *Cost* metric. Higher and further left is better.

Table 6.2 shows a *Cost* range, and the number (and percentage) of subject programs employing floating-point computations and continuous stochastic distributions for each debugger that incur the specified *Cost*. Figure 6.2 provides a graphical view of this data. In Figure 6.2 the x-axis represents the lower bound of each *Cost* range and the y-axis represents the percentage of programs where a *Cost* less than or equal to the upper bound is incurred.

Two trends are evident in this evaluation. The first is the profound incapability of CBI and Tarantula to localize faults in software employing floating-point computations and stochastic distributions. Here, neither approach is significantly better than the simple-minded fault localization approach described in Section 5.4.3.

Table 6.2: Number (percentage) of faulty version ranked statement lists in each score range for all approaches for the programs which employ floating-point computations and continuous stochastic distributions.

Cost	Tarantula # (%) of Programs	ESP # (%) of Programs	CBI # (%) of Programs
0% - 1%	2 (2.99%)	6 (8.96%)	16 (23.88%)
1% - 10%	10 (14.93%)	11 (16.42%)	25 (37.31%)
10% - 20%	8 (11.94%)	9 (13.43%)	9 (13.43%)
20% - 30%	6 (8.96%)	8 (11.94%)	4 (5.97%)
30% - 40%	9 (13.43%)	9 (13.43%)	3 (4.48%)
40% - 50%	6 (8.96%)	6 (8.96%)	2 (2.99%)
50% - 60%	7 (10.45%)	7 (10.45%)	2 (2.99%)
60% - 70%	8 (11.94%)	4 (5.97%)	1 (1.49%)
70% - 80%	5 (7.46%)	3 (4.48%)	2 (2.99%)
80% - 90%	3 (4.48%)	2 (2.99%)	1 (1.49%)
90% - 100%	3 (4.48%)	2 (2.99%)	2 (2.99%)

Table 6.3: Mean and median *Cost* for different subject program sizes.

Subject Program Size	# of Faulty Versions	Mean Cost %	Median Cost %
# of statements <1,000	292	18.66%	(8.32%)
1,000 <# of statements <10,000	41	14.24%	5.31%
10,000 <# of statements <100,000	54	11.64%	4.02%

Recall, this approach simply takes all statements included in failing test cases and ranks them in a random order. It is expected to localize faults in the same percentage of faulty versions as the *Cost* incurred. ESP is clearly more effective than this approach, as well as CBI and Tarantula.

The second trend is that ESP is consistently effective across a broad range of program types. While the original research objectives of this dissertation addressed effective fault localization analysis for programs with floating-point computations and continuous stochastic distributions, the outcome has been a fault localization approach that is consistently the best statistical debugger for all program types, including those originally targeted. This unexpected outcome means, among other things, that users whose sole concern is localizing faults while incurring minimal *Cost* do not need to choose between competing statistical debugging tools depending on subject program type. ESP is consistently their best choice.

While the inclusion of floating-point computations and continuous stochastic distributions does not significantly affect the fault localization capabilities of ESP, the size of a program does. Specifically, ESP incurs less *Cost* and thus becomes more effective when more statements are included in a subject program. This trend of improved effectiveness in larger programs is shown in Table 6.3.

It is not unexpected that ESP becomes more effective as more statements are included in a subject program. Additional statements generally create more control flow paths which enhances the fault localization capabilities of all the fault localization approaches included in our evaluation [130]. Furthermore, each of the faulty program versions where missing code qualifies as the fault include less than 1,000 statements.

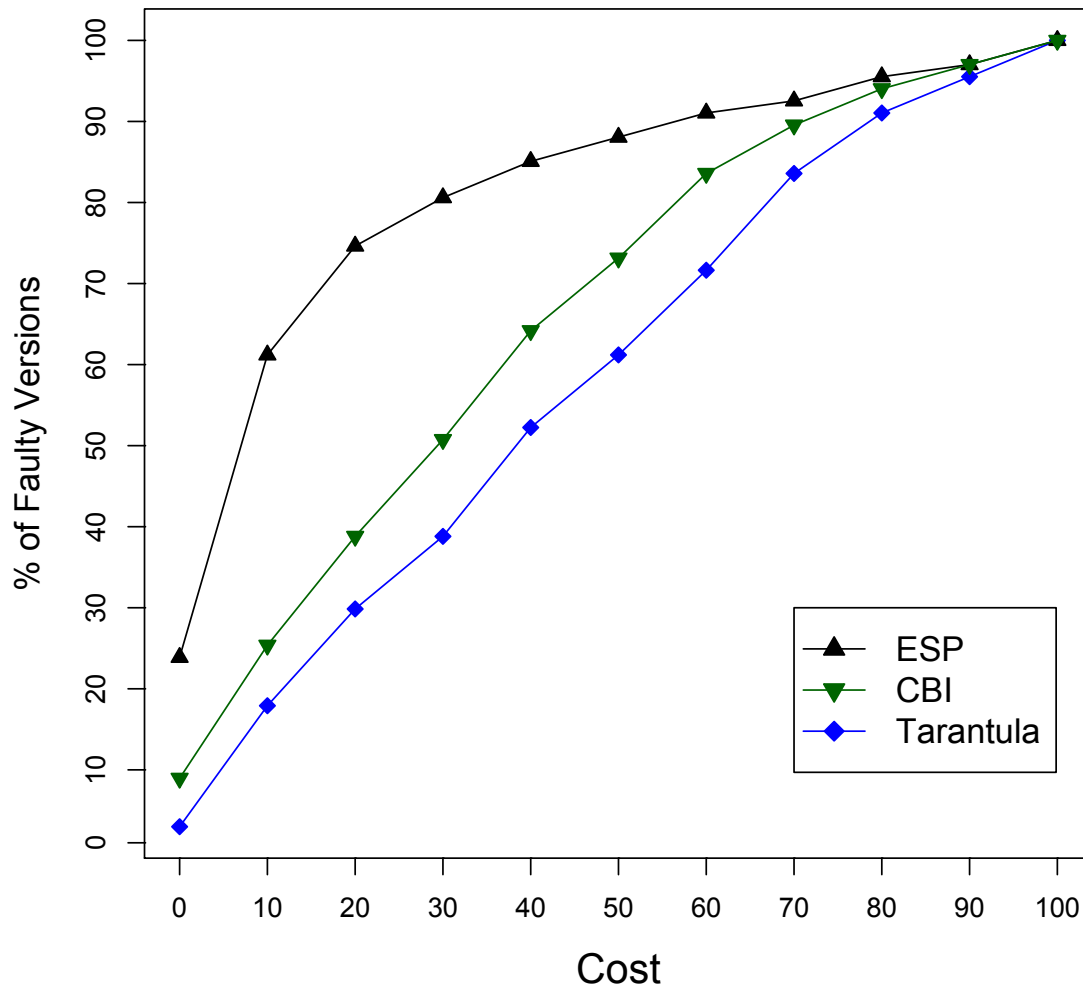


Figure 6.2: Comparison of the effectiveness of Tarantula, CBI and ESP using the *Cost* metric for the programs included in the evaluation which employ floating-point computations and continuous stochastic distributions. Higher and further left is better.

Recall, missing code faults violate the *coverage trigger assumption* in ESP, resulting in poor fault localization capabilities for these program versions. ESP assumes that the coverage of a statement corresponding to predicate p will necessarily trigger a failure. When this assumption is violated, as is the case of missing code faults, the resulting rankings from ESP are significantly less useful to developers.

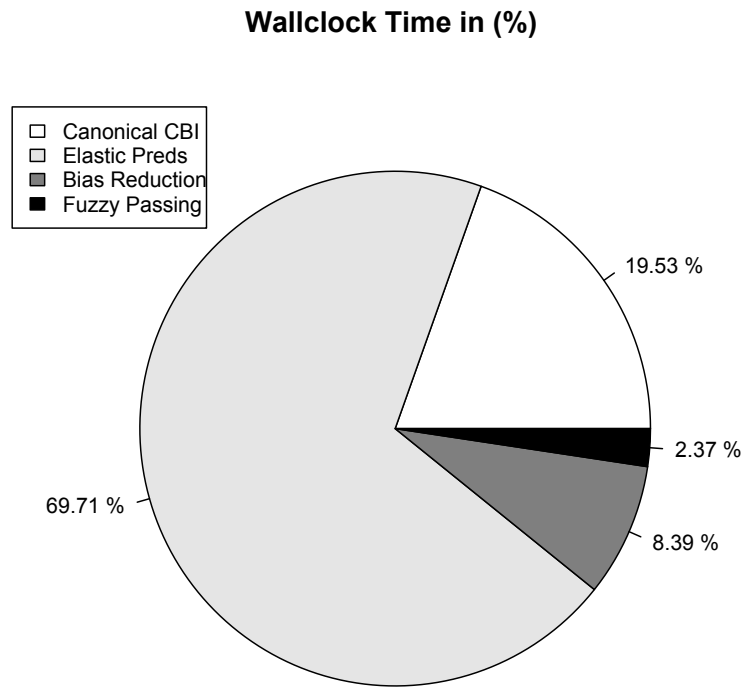


Figure 6.3: Graph of wallclock time overhead (in %) incurred by CBI when employing each contribution in this dissertation.

6.2 Overhead

The effectiveness evaluation in Section 6.1 shows that the fault localization capabilities of ESP are superior to those of Tarantula and CBI. This additional effectiveness is due to our novel and complimentary contributions presented in this dissertation. However, each of these contributions reduces the efficiency of ESP compared to CBI and Tarantula. The result is a significant increase in the time required by the debugger to return the list of statements (or predicates) in descending order of suspiciousness. For example, the itemized increase in execution time for ESP for the nine faulty versions of the adapted subject program `schedule` described in Section 5.4 is shown in Figure 6.3. Here, the wallclock time required to return the list of statements in ESP is five times more than CBI. This example program analysis reflects the average breakdown of wallclock time required by ESP to produce a ranked list of program statements. Recall, ESP reflects a version of CBI that employs each contribution in this dissertation.

Ideally the wallclock time incurred by each research contribution within ESP would be directly proportional to the improvements in fault localization effectiveness that it creates. However, within ESP the benefit of a research contribution is highly dependent on the subject program in which it is employed. The following list provides an analysis of the improvements in effectiveness offered by each of the three research contributions within ESP.

1. The elastic predicates within ESP offer the most improvement to the most programs. Of the 387 faulty program versions included in our evaluation, 253 incur less *Cost* when elastic predicates are employed as opposed to employing CBI alone. The mean reduction in *Cost* is 9.34%, the median reduction in *Cost* is 6.54% and the standard deviation in *Cost* is 7.54%. When evaluation is restricted to only the 67 faulty programs versions employing floating-point computations and continuous stochastic distributions, elastic predicates offer even more benefit. 55 of the 67 programs included in the evaluation which employ floating-point computations and continuous stochastic distributions incur less *Cost* when elastic predicates are employed as opposed to employing CBI alone. The mean reduction in *Cost* is 14.53%, the median reduction in *Cost* is 8.02% and the standard deviation in *Cost* is 13.83%.
2. The reduced bias suspiciousness estimation within ESP targets all software, not just software which employs floating-point computations and continuous stochastic distributions. As a result the reduced bias estimate offers improvements to a large portion of the subject programs included in our evaluation. When CBI is employed with our reduced bias suspiciousness estimate, the *Cost* of localizing faults in 241 of the 387 programs included in our evaluation is reduced. The mean reduction in *Cost* is 6.54%, the median reduction in *Cost* is 4.35% and the standard deviation in *Cost* is 6.83%. While these improvements in effectiveness are not as large as those achieved by elastic predicates they are still significant and affect a substantial portion of the subject programs included in our evaluation.
3. The concept of fuzzy passing extents is the most specific of the research contributions to a subject program. For fuzzy passing extents to offer improvement, a subject program *must* have variation in its output and the developer *must* be able to specify a *passing function* for the program. 59 of the 387 faulty program versions used in our evaluation meet these restrictions. While only a fraction of our evaluation programs meet these restrictions, those programs do reap more benefit from fuzzy passing extents than any of our other research contributions. When CBI is employed with fuzzy passing extents, the *Cost* of localizing faults in 57 of the 59 qualifying program versions is reduced. The mean *Cost* reduction is 16.13%, the median *Cost* reduction is 18.92% and the standard deviation in *Cost* is 8.68%.

While the applicability and *Cost* reduction offered by each of the research contributions within ESP varies, the evaluations in chapters 3, 4 and 5 show that improvements offered by each research contribution are largely independent of one another. This independence offers the opportunity to improve the overall efficiency of ESP by separately reducing the overhead of each contribution. In future work we will explore methods to accomplish this while maintaining the effectiveness of each contribution. This is a priority for the generation of elastic predicates since Figure 6.3 shows their construction incurs significantly more wallclock time than any of our other research contributions.

6.3 Time Incurred

Given that ESP is more effective at localizing faults, but requires significantly more wallclock time, a measure of overall user effectiveness is needed to determine if our work is successful. The *Time Incurred* metric serves this purpose. *Time Incurred* measures the overall time developers incur localizing faults in the subject programs using different statistical debuggers. This metric is composed of the following:

1. *Ranked Statement Time (RST)* - The time required by the fault localization approach to return the ranked set of program statements.
2. The time required by the software developer to identify the fault once the ranked set of statements is provided.

It is difficult to measure (2) because more studies are needed to determine how developers use the statement rankings provided by statistical debuggers [114]. *Cost* is the established measure of how useful a list of ranked predicates or statements is to a developer. However, *Cost* is not measured in time. *Time Incurred* addresses this problem. It includes the measure *Time Per Statement of (TPS)*, which reflects the time required for a user to either: (1) identify a fault in a given statement or (2) determine a given statement is fault free. The *Time Incurred* formula, shown in Equation 6.1, accounts for the total time required by a developer employing a statistical debugger to localize a fault in a given subject program by combining *RST*, *Cost*, *TPS* and the number of statements ranked for a given subject program (*NumOfStmts*). It is important to note that the minimum *Time Incurred* that can be achieved by an approach is the *RST* for the approach.

$$TimeIncurred = RST + (Cost \times TPS \times NumOfStmts) \quad (6.1)$$

Since the *TPS* can range from developer to developer, we consider a range of values for the *TPS*. This range is chosen to reflect all the reasonable projections for the amount of time users spend analyzing each

Table 6.4: Number (percentage) of faulty version ranked statement lists in each *Time Incurred* range for all approaches for all programs included in the evaluation.

Time Incurred (hrs)	% of Statements	Tarantula # (%) of Programs	ESP # (%) of Programs	CBI # (%) of Programs
0 - 41	(0% - 1%)	14 (3.62%)	0 (0.00%)	0 (0.00%)
41 - 382	(1% - 10%)	89 (23.00%)	190 (49.09%)	157 (40.56%)
382 - 760	(10% - 20%)	72 (18.60%)	91 (23.51%)	73 (18.86%)
760 - 1139	(20% - 30%)	33 (8.53%)	11 (2.84%)	27 (6.97%)
1139 - 1517	(30% - 40%)	54 (13.95%)	27 (6.97%)	23 (5.94%)
1517 - 1896	(40% - 50%)	27 (6.98%)	15 (3.87%)	30 (7.75%)
1896 - 2274	(50% - 60%)	14 (3.62%)	10 (2.58%)	16 (4.13%)
2274 - 2653	(60% - 70%)	21 (5.43%)	4 (1.03%)	4 (1.03%)
2653 - 3031	(70% - 80%)	12 (3.10%)	14 (3.62%)	13 (3.36%)
3031 - 3410	(80% - 90%)	16 (4.13%)	5 (1.29%)	19 (4.90%)
3410 - 3788	(90% - 100%)	35 (9.04%)	17 (4.39%)	22 (5.68%)
3788 - 4167	(100% - 110%)	0 (0.00%)	3 (0.77%)	3 (0.77%)

Table 6.5: Number (percentage) of faulty ranked statement lists in each *Time Incurred* range for all approaches for the programs which employ floating-point computations and continuous stochastic distributions.

Time Incurred (hrs)	% of Statements	Tarantula # (%) of Programs	ESP # (%) of Programs	CBI # (%) of Programs
0 - 8	(0% - 1%)	2 (2.99%)	0 (0.00%)	0 (0.00%)
8 - 16	(1% - 10%)	10 (14.93%)	30 (44.78%)	15 (22.39%)
16 - 24	(10% - 20%)	8 (11.94%)	13 (19.4%)	8 (11.94%)
24 - 32	(20% - 30%)	6 (8.96%)	7 (10.45%)	9 (13.43%)
32 - 40	(30% - 40%)	9 (13.43%)	5 (7.46%)	8 (11.94%)
40 - 48	(40% - 50%)	6 (8.96%)	3 (4.48%)	8 (11.94%)
48 - 56	(50% - 60%)	7 (10.45%)	2 (2.99%)	5 (7.46%)
56 - 64	(60% - 70%)	8 (11.94%)	1 (1.49%)	5 (7.46%)
64 - 72	(70% - 80%)	5 (7.46%)	2 (2.99%)	4 (5.97%)
72 - 80	(80% - 90%)	3 (4.48%)	1 (1.49%)	2 (2.99%)
80 - 88	(90% - 100%)	3 (4.48%)	2 (2.99%)	2 (2.99%)
88 - 96	(100% - 110%)	0 (0.00%)	1 (1.49%)	1 (1.49%)

statement ranked by statistical debuggers to determine if the statement contains a fault. It is important to note that the *TPS* is uniform for all statements in all subject programs. In this sense the *TPS* can be considered the mean time spent by a developer examining each statement in each subject program.

The work in this dissertation is successful if the predicate-level statistical debugger employing each of our contributions, ESP, localizes more faults than competing approaches at any given *Time Incurred*, for all values of *TPS* beginning at one minute (or 60 seconds). One minute (or 60 seconds) is chosen here to reflect the minimum time a developer could be expected to either: (1) identify a fault in a given statement or (2) determine a given statement is fault free. Other *TPS* below one minute are also discussed in this portion of the evaluation.

The *Time Incurred* for Tarantula, CBI and ESP for the 31 subject programs and 387 faulty program versions is shown in Table 6.4. Similarly, the *Time Incurred* for the 11 subject programs and 67 faulty

program versions employing floating-point computations and continuous stochastic distributions is shown in Table 6.5. Each row in Table 6.4 and Table 6.5 shows a *Time Incurred* range and how many faults each approach localized within the time range. The ranges are calculated by: (1) summing the *Time Incurred* to localize the fault for each faulty program version using Tarantula and (2) dividing that sum into 1%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 100% and 110% *Time Incurred* ranges. Tarantula is used as the basis for each *Time Incurred* range because it incurs the least time to: (1) find the fault in one of the faulty program versions and (2) find the fault in all of the faulty program versions included in our evaluation. The *Time Incurred* range in the bottom row in tables 6.4 and 6.5 (100%-110%) reflects the time required to localize faults in all of the faulty program versions included in our evaluation using ESP and CBI.

Figures 6.4 and 6.5 provide a graphical view of the data in Table 6.4 and Table 6.5. In figures 6.4 and 6.5 the x-axis represents the lower bound of each *Time Incurred* range and the y-axis represents the percentage of subject programs where the *Time Incurred* is less than or equal to the upper bound. This presentation of data follows the established convention in the fault localization community which employs *Cost* along the x-axis instead of *Time Incurred* [7, 12, 22, 23].

Recall in figures 6.1 and 6.2, given a specified *Cost* range, ESP localized more faults than any of the competing debuggers. Similarly in figures 6.4 and 6.5, in the preponderance of *Time Incurred* ranges, ESP localizes more faults than its competitors. However, in two different *Time Incurred* ranges, Tarantula localizes more faults than ESP or CBI. We explore Tarantula's performance in these *Time Incurred* ranges next.

Throughout our evaluations Tarantula has been the most efficient statistical debugging approach. As a result, in the evaluation shown in tables 6.4 and 6.5 and figures 6.4 and 6.5 there exist two *Time Incurred* ranges where Tarantula localizes more faults than ESP or CBI. The first period occurs at the beginning of a fault localization analysis, where Tarantula has generated a ranked list of statements and ESP and CBI have not finished generating such a list. This circumstance enables a developer using Tarantula to start identifying faults, while a user employing CBI or ESP cannot. The second period occurs at the end of the fault localization analysis, where a developer employing Tarantula has searched through all the ranked statements and thus identified all the faults. In this case, developers employing ESP and CBI have not had sufficient time to search all of the ranked statements and thus have not identified all of the faults. These are the only two ranges of *Time Incurred* where Tarantula outperforms ESP or CBI. In all of the other time ranges ESP localizes more faults than CBI or Tarantula.

The superior performance of ESP holds for *TPS* values greater than seven seconds, except for the extreme conditions already noted. While our evaluation begins with an already short assumption of one minute, we could have chosen an arguably unrealistic value as small as seven seconds and still claimed superiority. For *TPS* values smaller than seven seconds ESP is unable to produce a ranked list of statements before a user

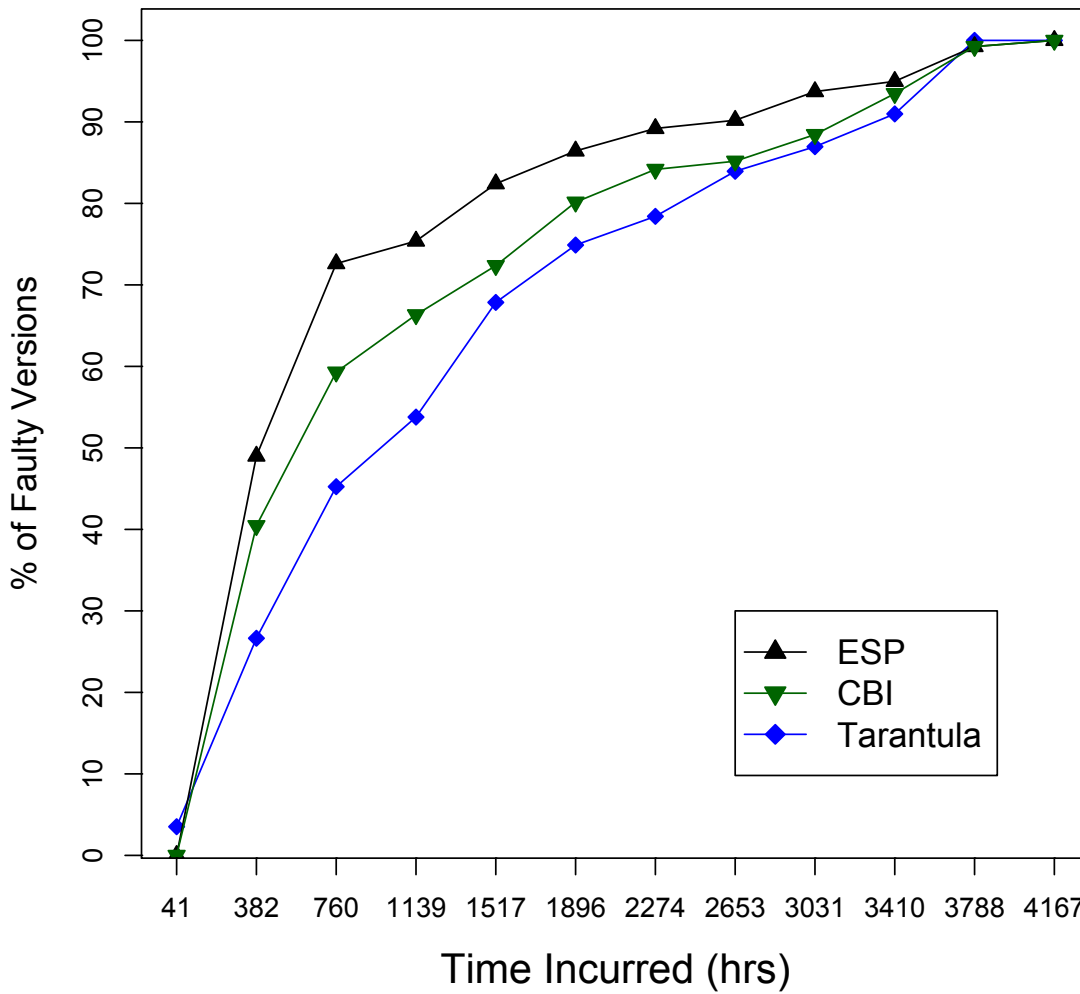


Figure 6.4: Comparison of the effectiveness of Tarantula, CBI and ESP using the *Time Incurred* metric. Higher and further left is better.

employing Tarantula has explored 10% of the ranked list of statements. For these *TPS* values Tarantula and CBI are capable of localizing more faults than ESP in at least three different, albeit narrow, *Time Incurred* ranges. For *TPS* values of zero, ESP is outperformed by Tarantula and CBI in every *Time Incurred* range because only the efficiency of a fault localization approach is considered, not the effectiveness.

Given that Tarantula is not significantly more effective than ESP in the two time ranges where it outperforms ESP and its superior effectiveness is owed more to efficiency than analysis capabilities, we conclude that our novel contributions featured in ESP are successful. Furthermore, the focus of the implementation of each of the research contributions within ESP is effectiveness, not efficiency. In the

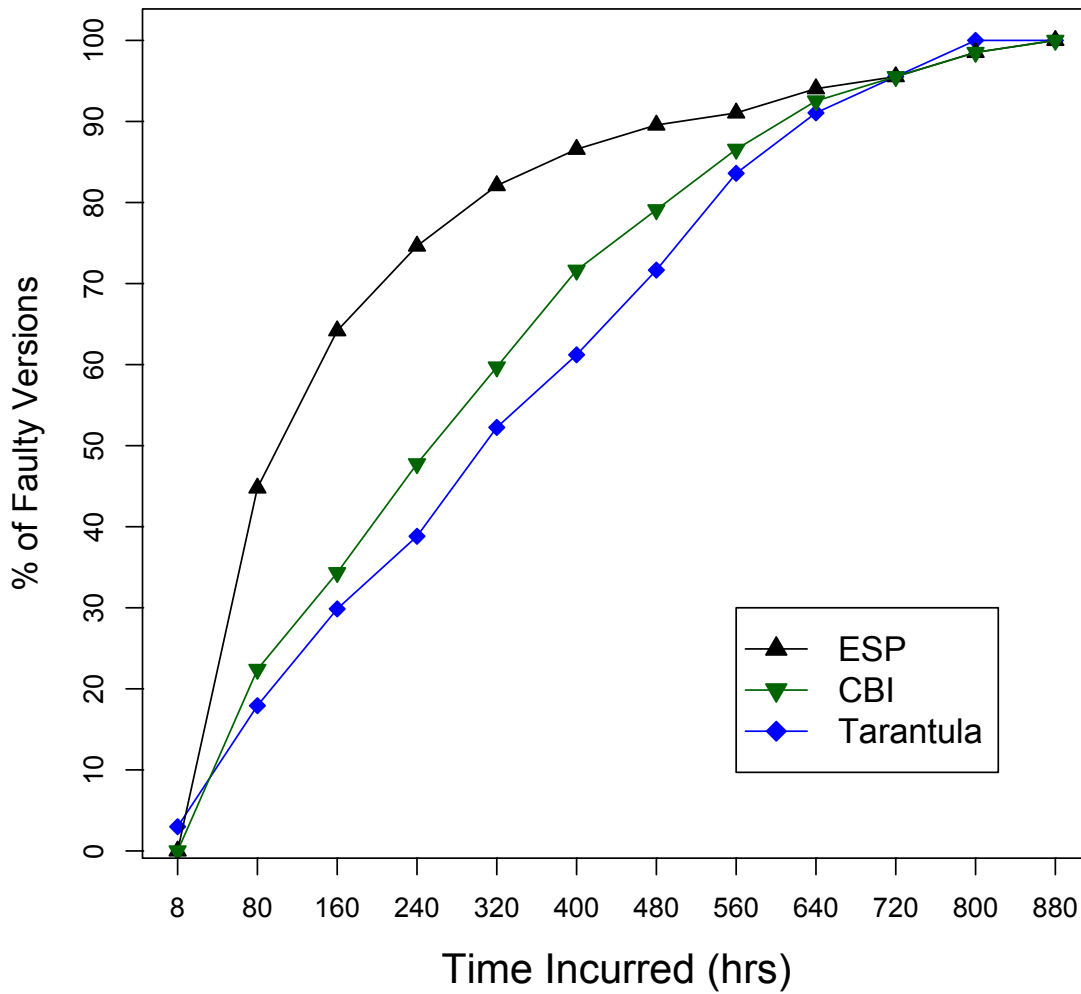


Figure 6.5: Comparison of the effectiveness of Tarantula, CBI and ESP using the *Time Incurred* metric for the programs included in the evaluation which employ floating-point computations and continuous stochastic distributions. Higher and further left is better.

future we expect efficiency improvements simply from honing the implementation of each of these research contributions. Ultimately, the research contributions within ESP enhance the effectiveness of automated debuggers and are a valuable addition to a software engineer’s toolset, especially for programs employing floating-point computations or continuous stochastic distributions.

6.4 Summary

The research contributions presented in chapters 3, 4 and 5 enable developers to localize faults causing failures more effectively than existing alternatives in general purpose software and programs employing floating-point and continuous stochastic distributions. While the applicability and effectiveness offered by each of our research contributions varies, the analysis and discussion in this chapter show that improvements offered by each research contribution are largely independent of one another. Furthermore, despite the additional overhead incurred by each of these research objectives, the *Time Incurred* metric demonstrates that these contributions significantly reduce the overall time developers incur localizing faults in a subject program compared to existing alternatives.

Chapter 7

Conclusion

When a failure is first observed in software, the prospect of localizing the fault causing it can be daunting. Common practice is to apply classic debugging techniques to identify the program statements and interactions that lead to the fault. This practice is largely manual and it can consume weeks, months and even years of effort. Resources in terms of money, people and time should be minimized in the process of fault localization. Predicate-level statistical debuggers have been employed to meet this goal.

While predicate-level statistical debuggers are effective in general, they are not tailored to an important class of software, including simulations and computational models, which employ floating-point computations and continuous stochastic distributions. In this dissertation that deficiency, and coincidentally, fault localization for all program types, is addressed through three novel and complementary contributions. Here, a summary of these contributions is presented and a description of the availability of the tool, ESP, which implements them is provided. New research areas and opportunities for future work stemming from these contributions are also reviewed.

7.1 Summary of Contributions

Three contributions presented in this dissertation enable users to more effectively isolate causes of failures in software, in particular in software employing floating-point computations and continuous stochastic distributions. A summary of how these contributions interact with one another is presented here.

1. Elastic predicates are formed from summary statistics of variable values profiled at instrumented program points. These predicates more closely match where a fault is expressed and lead to more

effective fault localization, especially for software employing floating-point computations and continuous stochastic distributions.

2. The metrics employed to estimate the suspiciousness of predicates in statistical debuggers (including elastic predicates) do not account for several factors which significantly influence how frequently some predicates appear in passing and failing test cases. The inability of existing estimates to control for these factors reflects confounding bias in the suspiciousness estimate. Casting predicate-level statistical debugging as an observational study and employing causal models and statistical matching significantly reduces or eliminates this bias and yields more accurate suspiciousness estimation.
3. Suspiciousness estimation requires a distinction between test cases which produce the specified output (and pass) versus those test cases which produce unspecified output (and fail). Current statistical debuggers have not assumed any variance from the specified (or passing) output for a given test case. This assumption precludes the practical consideration of a large class of software, including exploratory simulations, which employ stochastics to reflect uncertainty in an underlying model.

This restriction can be lifted through fuzzy passing extents which enable users to define a function which specifies the degree ($[0, 1]$) to which a faulty program passes a given test case. For these programs fuzzy passing extents can significantly improve the effectiveness of existing statistical debuggers compared to their traditional counterparts which do not employ fuzzy passing extents. Furthermore, they can reproduce the results of the suspiciousness estimates presented in this dissertation if needed. This ensures that debuggers employing fuzzy passing extents will not be outperformed by their traditional counterparts which do not employ fuzzy passing extents.

Ultimately, the research objectives within ESP have changed the landscape of predicate-level statistical debugging. The combination of elastic predicates, reduced bias suspiciousness estimation and fuzzy passing extents makes ESP more effective and applicable than existing predicate-level statistical debugging approaches.

In Chapter 6 we showed that ESP significantly reduces the overall time developers spend localizing faults compared to the competing statistical debuggers. However, in terms of the wallclock time required to produce statement rankings ESP is generally incapable of outperforming statement-level statistical debugging approaches, such as Tarantula. As things stand, there is a trade-off between ESP's effectiveness and Tarantula's efficiency. Assuming a base-line level of fault localization effectiveness, the chief concern for *some* developers is the wallclock time required to produce statement rankings [114]. In these cases ESP's ability to produce statement rankings efficiently (relatively speaking) can be an issue. As a result, despite the superior fault localization capabilities of ESP, continued improvements to produce reasonably effective rankings as efficiently as possible for statement-level debuggers are strongly desirable.

It is also important to note where ESP can be bested in effectiveness. The evaluation of elastic predicates in Chapter 3 showed that for subject programs that do not employ floating-point computations *or* continuous stochastic distributions, state-altering approaches, such as IVMP, can outperform ESP. Furthermore, the evaluations in chapters 4 and 5 show that ESP can be fooled by missing code. Recently, there has been some success in addressing these faults from approaches which insert mutations of suspicious predicates into faulty subject programs in an attempt to make failing test cases become passing test cases [131]. Exploring the improvements in effectiveness and efficiency that our research contributions can offer these approaches is a new research area opened by ESP. This and several other new research areas are explored next.

7.2 New Research Areas

The research and evaluations within this dissertation shed light on several new areas of fault localization research worthy of pursuit. Here, each of these areas is reviewed. These areas differ from the future avenues of research presented in Section 7.4 in breadth and scope. In contrast, the avenues for future work presented in Section 7.4 are more narrow and are directly tied to improving the capabilities of ESP.

One research opportunity made apparent by our work is improvement of the fault localization capabilities of state-altering debuggers for subject programs employing floating-point computations and stochastics. The evaluation of IVMP in Chapter 3 is the first identification of the limitations of state-altering approaches for these types of subject programs [26]. This deficiency presents an opportunity to develop a state-altering analysis that does not rely on *repeatable execution traces* and *exact output matching*. Addressing this deficiency could advance the state of the art in fault localization for software employing floating-point computations and stochastics.

New suspiciousness estimates for predicate-level statistical debuggers are also needed in the field of fault localization. The causal model introduced in Chapter 4 reduces the confounding bias in existing suspiciousness estimates but it does not necessarily eliminate it. Fundamentally different predicate-level metrics that consider more than just the presence of a predicate in a passing and failing test case need to be explored to identify more effective suspiciousness measures which are less susceptible to bias.

Finally, our work has demonstrated the need to explore how useful developers find the statement rankings provided by statistical debuggers. Recall, it is currently unclear how useful developers find ranking-based approaches to fault localization [114]. Answers need to be provided to questions such as: (1) Does the effectiveness of an automated debugger increase with the level of difficulty of the debugging task?, (2) Are ranked lists of statements (or predicates) the best way to relate analysis back to a fault-searching developer? and (3) How difficult is it for developers to choose fuzzy passing functions, in subject programs with variance

in the output? It is important to note that the goal of this research is not to demonstrate that a particular approach is better than another one, rather it is to gather insight on how to build better automated debugging tools and identify promising research directions in fault localization.

7.3 Availability of ESP

A version of ESP which employs elastic predicates and fuzzy passing extents is publicly available [132]. The downloadable content includes the source code, build instructions and an example faulty subject program on which ESP can be employed. Recently, a University of Virginia undergraduate student working with a queueing simulation successfully downloaded and localized several faults in a queueing simulation with this version of ESP. In the future a full version of ESP which includes our reduced bias suspiciousness estimation will be made available.

7.4 Future Work

Several future directions of research directly related to our research contributions warrant further pursuit. These opportunities for future work are described next.

1. **Efficient generation of maximized elastic predicates.** Elastic predicates are a significant improvement over existing predicates because they use summary statistics to adjust to the values of program variables. However, the summary statistics used in these predicates do not maximize the suspiciousness of the predicate. Recall from Chapter 3, that further improvement is possible by generating a *maximized* predicate where the predicate endpoint value maximizes the suspiciousness score for the predicate. Accomplishing this functionality, or even approximating it, efficiently is a research challenge.
2. **Further reducing confounding bias in predicate-level suspiciousness estimates by controlling for data flow dependencies.** Recall from Chapter 4 that subject program control flow dependencies and statement coverage can produce significant confounding bias in established predicate-level suspiciousness estimates. Subject program data flow dependencies also contribute to confounding bias in these estimates. Three research challenges arise as a result: (1) profiling subject programs to identify these dependencies, (2) incorporating them as covariates into our causal model(s) and (3) matching test cases in the face of additional covariates. Overcoming these challenges will continue to reduce and/or eliminate confounding bias in suspiciousness estimates and result in more effective fault localization.

3. **Focused elastic predicate instrumentation.** Most instrumented predicates are poor predictors of failure. A focused monitoring strategy that mitigates instrumenting static predicates that are poor failure predictors has been proposed [14]. This strategy improves the efficiency of the fault localization process but not the effectiveness. The advent of elastic predicates in this dissertation requires a similar strategy. Developing a methodology to mitigate instrumenting elastic predicates which are poor failure predictors is an avenue for future work. Addressing this research goal will allow users to employ full elastic predicate instrumentation when efficiency is not an issue and specify the degree to which elastic predicate instrumentation should be reduced when efficiency is an issue. This future work will enable users to balance effectiveness and efficiency.
4. **Parallelization.** The suspiciousness estimation of each predicate instrumented in a subject program is computed in isolation from all other predicates. Thus, multiple cores can carry out predicate suspiciousness estimation in parallel. Similarly, the execution of each test case required to compute summary statistics needed to generate elastic predicates is an independent process. These test cases can also be executed in parallel. The efficiency of statistical debuggers employing our contributions can be improved significantly by taking advantage of these embarrassingly parallel properties. The engineering challenge of creating such a framework reflects an opportunity to significantly reduce the *Time Incurred* for developers who employ ESP.
5. **Reusing data when localizing multiple faults.** Localizing multiple faults in a subject program is an incremental process. Recall, in Section 3.4.11 an algorithm is presented that ensures that at least one fault localizing predicate will be produced for each fault in a subject program. While this algorithm is effective, it is not optimized for efficiency. Specifically, no information is reused between the ranked list of predicates produced for each fault. In future work, we will explore how reusing information between iterations of the multiple fault localization algorithm can improve the efficiency of ESP in terms of time and space.

7.5 Final Remarks

The contributions presented in this dissertation advance the state of the art in localizing sources of failures in software. Elastic predicates are instrumented in faulty subject programs to provide an adaptive and more granular approach to capturing the conditions where a fault is and is not expressed. Accounting for the control flow dependencies between predicates and statement coverage with statistical matching and causal models enables the confounding bias in existing predicate suspiciousness estimation metrics to be significantly

Table 7.1: Publications featuring the contributions of this dissertation.

Chapter	Venue	Publication
3	ASE '11	<i>Statistical Debugging with Elastic Predicates</i> [26]
4	ICSE '12	<i>Reducing Bias In Predicate-Level Statistical Debugging Metrics</i> [33]
4	NFM '12	<i>Modifying Test Suites to Enable Effective Predicate-Level Statistical Debugging</i> [34]
5	WSC '11	<i>Applying Enhanced Fault Localization Technology to Monte Carlo Simulations</i> [128] (Nominated for Best Conference Paper)

reduced and/or eliminated. Finally, fuzzy passing extents enable elastic predicates, reduced confounding bias estimates, and a bevy of other fault localization contributions to be applied to the programs which fault localization was previously inapplicable, because of variance in the program output.

Table [7.1](#) shows the major software engineering and simulation research venues where these contributions have been published. We expect that the contributions will be adopted in existing statistical debuggers and they will contribute to more effective localization of the sources of failing outcomes, especially for developers and users of software with floating-point computations and continuous stochastic distributions.

Bibliography

- [1] John C. Munson, Allen P. Nikora, and Joseph S. Sherif. Software faults: a quantifiable definition. *Adv. Eng. Softw.*, 37(5):327–333, May 2006.
- [2] J.C. C. Laprie, A. Avizienis, and H. Kopetz, editors. *Dependability: Basic Concepts and Terminology*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.
- [3] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *SOFTWARE TESTING, VERIFICATION AND RELIABILITY*, 10:2000, 2000.
- [4] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. *SIGPLAN Not.*, 33:83–90, July 1998.
- [5] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada*, ASE '03, pages 30–39. IEEE Computer Society, 2003.
- [6] I Vessey. Expertise in debugging computer programs: an analysis of the content of verbal protocols. *IEEE Trans. Syst. Man Cybern.*, 16:621–637, September 1986.
- [7] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated software engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.
- [8] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM.
- [9] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. *SIGPLAN Not.*, 38:141–154, May 2003.
- [10] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming language design and implementation*, PLDI '05, pages 15–26, New York, NY, USA, 2005. ACM.
- [11] Ben Liblit. Cooperative debugging with five hundred million test cases. In *Proceedings of the 2008 International Symposium on Software testing and analysis*, ISSTA '08, pages 109–120, New York, NY, USA, 2008. ACM.
- [12] Rui Abreu, Peter Zoetewij, and Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 89–98, Washington, DC, USA, 2007. IEEE Computer Society.
- [13] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. Sober: statistical model-based bug localization. *SIGSOFT Softw. Eng. Notes*, 30:286–295, September 2005.

- [14] Piramanayagam Arumuga Nainar and Ben Liblit. Adaptive bug isolation. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 255–264, New York, NY, USA, 2010. ACM.
- [15] Piramanayagam Arumuga Nainar, Ting Chen, Jake Rosin, and Ben Liblit. Statistical debugging using compound boolean predicates. In *Proceedings of the 2007 International Symposium on Software testing and analysis*, ISSTA '07, pages 5–15, New York, NY, USA, 2007. ACM.
- [16] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 34–44, Washington, DC, USA, 2009. IEEE Computer Society.
- [17] Zhenyu Zhang, Bo Jiang, W. K. Chan, T. H. Tse, and Xinming Wang. Fault localization through evaluation sequences. *J. Syst. Softw.*, 83:174–187, February 2010.
- [18] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of software engineering*, SIGSOFT '02/FSE-10, pages 1–10, New York, NY, USA, 2002. ACM.
- [19] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 28:2002, 2002.
- [20] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software engineering*, ICSE '05, pages 342–351, New York, NY, USA, 2005. ACM.
- [21] Ghassan Mishherghi and Zhendong Su. Hdd: hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software engineering*, ICSE '06, pages 142–151, New York, NY, USA, 2006. ACM.
- [22] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. Fault localization using value replacement. In *Proceedings of the 2008 International Symposium on Software testing and analysis*, ISSTA '08, pages 167–178, New York, NY, USA, 2008. ACM.
- [23] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. Effective and efficient localization of multiple faults using value replacement. In *25th IEEE International Conference on Software Maintenance*, ICSM '09, pages 221–230, 2009.
- [24] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In *Proceedings of the 28th International Conference on Software engineering*, ICSE '06, pages 272–281, New York, NY, USA, 2006. ACM.
- [25] American Psychological Association. *Publication manual of the American Psychological Association*. American Psychological Association, 2001.
- [26] Ross Gore, Paul F. Reynolds Jr., and David Kamensky. Statistical debugging with elastic predicates. In *Proceedings of the 26th IEEE/ACM International Conference on Automated software engineering*, ASE '11, pages 492–495, New York, NY, USA, 2011. ACM.
- [27] Yanbing Yu, James A. Jones, and Mary Jean Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of the 30th International Conference on Software engineering*, ICSE '08, pages 201–210, New York, NY, USA, 2008. ACM.
- [28] James A. Jones, James F. Bowring, and Mary Jean Harrold. Debugging in parallel. In *Proceedings of the 2007 International Symposium on Software testing and analysis*, ISSTA '07, pages 16–26, New York, NY, USA, 2007. ACM.

- [29] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. In *Proceedings of the 28th International Conference on Software engineering*, ICSE '06, pages 82–91, New York, NY, USA, 2006. ACM.
- [30] Zhenyu Zhang, W. K. Chan, T. H. Tse, Bo Jiang, and Xinming Wang. Capturing propagation of infected program states. In *Proceedings of the the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 43–52, New York, NY, USA, 2009. ACM.
- [31] George K. Baah, Andy Podgurski, and Mary Jean Harrold. Causal inference for statistical fault localization. In *Proceedings of the 19th International Symposium on Software testing and analysis*, ISSTA '10, pages 73–84, New York, NY, USA, 2010. ACM.
- [32] George K. Baah, Andy Podgurski, and Mary Jean Harrold. Mitigating the confounding effects of program dependences for effective fault localization. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 146–156, New York, NY, USA, 2011. ACM.
- [33] Ross J. Gore and Paul F. Reynolds Jr. Reducing bias in predicate-level suspiciousness metrics. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 34–44, Zurich, Switzerland, 2012. IEEE Computer Society.
- [34] Ross Gore and Paul F. Reynolds Jr. Modifying test suite composition to enable effective predicate-level statistical debugging. In Alwyn Goodloe and Suzette Person, editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 70–84. Springer Berlin / Heidelberg, 2012.
- [35] Mark Weiser. Programmers use slices when debugging. *Commun. ACM*, 25:446–452, July 1982.
- [36] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM.
- [37] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9:319–349, July 1987.
- [38] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19:177–184, April 1984.
- [39] T. Reps and T. Bricker. Illustrating interference in interfering versions of programs. In *Proceedings of the 2nd International Workshop on Software configuration management*, SCM '89, pages 46–55, New York, NY, USA, 1989. ACM.
- [40] Bogdan Korel and Jurgen Rilling. Application of dynamic slicing in program debugging. In *Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG '97)*, Linköping, 1997.
- [41] Bogdan Korel and Satish Yalamanchili. Forward computation of dynamic program slices. In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software testing and analysis*, ISSTA '94, pages 66–79, New York, NY, USA, 1994. ACM.
- [42] Bogdan Korel and Jurgen Rilling. Dynamic program slicing in understanding of program execution. In *Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*, WPC '97, pages 80–, Washington, DC, USA, 1997. IEEE Computer Society.
- [43] Mariam Kamkar, Nahid Shahmehri, and Peter Fritzson. Interprocedural dynamic slicing. In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 370–384, London, UK, 1992. Springer-Verlag.
- [44] G. A. Venkatesh. The semantic approach to program slicing. *SIGPLAN Not.*, 26:107–119, May 1991.

- [45] Jim Q. Ning, Andre Engberts, and W. Voytek Kozaczynski. Automated support for legacy code understanding. *Commun. ACM*, 37:50–57, May 1994.
- [46] John Field and Frank Tip. Dynamic dependence in term rewriting systems and its application to program slicing. In *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming*, pages 415–431, London, UK, 1994. Springer-Verlag.
- [47] Xiangyu Zhang and Rajiv Gupta. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software engineering*, ICSE '03, pages 319–329, 2003.
- [48] Xiangyu Zhang and Rajiv Gupta. Cost effective dynamic program slicing. *SIGPLAN Not.*, 39:94–106, June 2004.
- [49] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. Critical slicing for software fault localization. *SIGSOFT Softw. Eng. Notes*, 21:121–134, May 1996.
- [50] Hiralal Agrawal, Richard A. Demillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Softw. Pract. Exper.*, 23:589–616, June 1993.
- [51] Hiralal Agrawal, Joseph R. Horgan, Saul London, and W. Eric Wong. Fault localization using execution slices and dataflow tests. In *Proceedings of the Sixth International Symposium on Software Reliability Engineering*, ISSRE '95, pages 143–151, 1995.
- [52] Peter Fritzson, Nahid Shahmehri, Mariam Kamkar, and Tibor Gyimothy. Generalized algorithmic debugging and testing. *ACM Lett. Program. Lang. Syst.*, 1:303–322, December 1992.
- [53] Jens Krinke. Visualization of program dependence and slices. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 168–177, Washington, DC, USA, 2004. IEEE Computer Society.
- [54] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. Experimental evaluation of using dynamic slices for fault location. In *Proceedings of the sixth International Symposium on Automated analysis-driven debugging*, AADeBUG'05, pages 33–42, New York, NY, USA, 2005. ACM.
- [55] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Pruning dynamic slices with confidence. *SIGPLAN Not.*, 41:169–180, June 2006.
- [56] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27:99–123, February 2001.
- [57] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 291–301, New York, NY, USA, 2002. ACM.
- [58] Pin Zhou, Wei Liu, Long Fei, Shan Lu, Feng Qin, Yuanyuan Zhou, Samuel Midkiff, and Josep Torrellas. Accmon: Automatically detecting memory-related bugs via program counter-based invariants. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 269–280, Washington, DC, USA, 2004. IEEE Computer Society.
- [59] Christoph Csallner and Yannis Smaragdakis. Check 'n' crash: combining static checking and testing. In *Proceedings of the 27th International Conference on Software engineering*, ICSE '05, pages 422–431, New York, NY, USA, 2005. ACM.
- [60] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, pages 504–527, Glasgow, Scotland, July 27–29, 2005.
- [61] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39:92–106, December 2004.

- [62] Shan Lu, Pin Zhou, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. Pathexpander: Architectural support for increasing the path coverage of dynamic bug detection. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 38–52, Washington, DC, USA, 2006. IEEE Computer Society.
- [63] Alex David Groce. *Error explanation and fault localization with distance metrics*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2005. AAI3166277.
- [64] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. *Int. J. Softw. Tools Technol. Transf.*, 8:229–247, June 2006.
- [65] Andrew J. Ko and Brad A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software engineering*, ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM.
- [66] Andrew J. Ko and Brad A. Myers. Finding causes of program output with the java whyline. In *Proceedings of the 27th International Conference on Human factors in computing systems*, CHI '09, pages 1569–1578, New York, NY, USA, 2009. ACM.
- [67] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.
- [68] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.
- [69] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, POPL '02, pages 128–139, New York, NY, USA, 2002. ACM.
- [70] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. Heapmon: a helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *IBM J. Res. Dev.*, 50:261–275, March 2006.
- [71] Feng Qin, Shan Lu, and Yuanyuan Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 291–302, Washington, DC, USA, 2005. IEEE Computer Society.
- [72] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with wit. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 263–277, Washington, DC, USA, 2008. IEEE Computer Society.
- [73] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *In Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, 2004.
- [74] Jun Xu and Nithin Nakka. Defeating memory corruption attacks via pointer taintedness detection. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, pages 378–387, Washington, DC, USA, 2005. IEEE Computer Society.
- [75] Junfeng Yang, Can Sar, and Dawson Engler. Explode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating systems design and implementation*, OSDI '06, pages 131–146, Berkeley, CA, USA, 2006. USENIX Association.
- [76] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32:176–192, March 2006.
- [77] David Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming language design and implementation*, PLDI '96, pages 44–53, New York, NY, USA, 1996. ACM.

- [78] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30:775–802, June 2000.
- [79] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. *SIGSOFT Softw. Eng. Notes*, 25:14–25, August 2000.
- [80] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop on Model checking of software*, SPIN '01, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.
- [81] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with blast. In *Proceedings of the 10th International Conference on Model checking software*, SPIN'03, pages 235–239, Berlin, Heidelberg, 2003. Springer-Verlag.
- [82] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 1–12, New York, NY, USA, 2002. ACM.
- [83] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28:1035–1087, November 2006.
- [84] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. Pse: explaining program failures via postmortem static analysis. *SIGSOFT Softw. Eng. Notes*, 29:63–72, October 2004.
- [85] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. *SIGPLAN Not.*, 40:351–363, January 2005.
- [86] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. *SIGPLAN Not.*, 37:234–245, May 2002.
- [87] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35:57–72, October 2001.
- [88] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. *SIGPLAN Not.*, 37:4–16, January 2002.
- [89] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European software engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of software engineering*, ESEC/FSE-13, pages 306–315, New York, NY, USA, 2005. ACM.
- [90] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *SIGSOFT Softw. Eng. Notes*, 30:296–305, September 2005.
- [91] Judea Pearl and Thomas S. Verma. A theory of inferred causation. In Brian Skyrms Dag Prawitz and Dag Westersth, editors, *Logic, Methodology and Philosophy of Science IX Proceedings of the Ninth International Congress of Logic, Methodology and Philosophy of Science*, volume 134 of *Studies in Logic and the Foundations of Mathematics*, pages 789 – 811. Elsevier, 1995.
- [92] Judea Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, March 2000.
- [93] B. Fristedt and L.F. Gray. *A Modern Approach to Probability Theory*. Probability and Its Applications. Birkhäuser, 1997.
- [94] Stephen L. Morgan and Christopher Winship. *Counterfactuals and Causal Inference: Methods and Principles for Social Research (Analytical Methods for Social Research)*. Cambridge University Press, 1 edition, July 2007.

- [95] G. Casella and R. L. Berger. *Statistical Inference*. Wadsworth and Brooks/Cole, Pacific Grove, CA, 2002.
- [96] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
- [97] bc calculator. <http://www.gnu.org/software/bc/>.
- [98] R.J. Herrnstein. *Bell Curve: Intelligence and Class Structure in American Life*. Free Press, New York, 1994.
- [99] W. Bryc. *The normal distribution: characterizations with applications*. Lecture notes in statistics. Springer-Verlag, 1995.
- [100] S. Ghahramani. *Fundamentals of probability with stochastic processes*. Pearson/Prentice Hall, 2005.
- [101] N. Balakrishnan and V.B. Nevzorov. *A primer on statistical distributions*. Wiley-interscience. Wiley, 2003.
- [102] G. McPherson. *Statistics in scientific investigation: its basis, application, and interpretation*. Springer texts in statistics. Springer-Verlag, 1990.
- [103] Fabio Pellacini. User-configurable automatic shader simplification. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 445–452, New York, NY, USA, 2005. ACM.
- [104] George K. Baah, Andy Podgurski, and Mary Jean Harrold. The probabilistic program dependence graph and its application to fault diagnosis. In *Proceedings of the 2008 International Symposium on Software testing and analysis*, ISSTA '08, pages 189–200, New York, NY, USA, 2008. ACM.
- [105] K. Detlefsen and Wolfgang K. Hrdle. Calibration risk for exotic options. *The Journal of Derivatives*, 14(4):47 – 63, 2007.
- [106] S. Mikhailov and U. Ngel. Heston's stochastic volatility model: Implementation, calibration, and some extensions. *Wilmott Magazine*, pages 74 –79, 2003.
- [107] quant code. <http://www.quantcode.com/>.
- [108] um-olsr: Optimized link state routing implemented for ns2. <http://masimum.dif.um.es/?Software:UM-OLSR>.
- [109] Bug in um-olsr? <http://stackoverflow.com/questions/4190561/bug-in-um-olsr-for-ns-2-34>.
- [110] ns2: The network simulator. <http://is.edu/nsam/ns>.
- [111] ns2 problems. <http://www.isi.edu/nsnam/ns/ns-problems.html>.
- [112] W. David Kelton. Representing and generating uncertainty effectively. In *Proceedings of the 39th Conference on Winter simulation: 40 years! The best is yet to come*, WSC '07, pages 38–42, Piscataway, NJ, USA, 2007. IEEE Press.
- [113] D. Gross, J.F. Shortle, J.M. Thompson, and C.M. Harris. *Fundamentals of Queueing Theory*. Wiley Series in Probability and Statistics. John Wiley & Sons, 2011.
- [114] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, New York, NY, USA, 2011. ACM.
- [115] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, 2002. Springer-Verlag.

- [116] N.A. Gershenfeld. *The Nature of Mathematical Modeling*. Cambridge University Press, 1999.
- [117] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.
- [118] Software-artifact infrastructure repository. <http://sir.unl.edu/portal/index.php>.
- [119] James K. Kuchar and Ann C. Drumm. The traffic alert and collision avoidance system. *Lincoln Laboratory Journal*, 16(2):277 – 296, 2007.
- [120] R. De Maesschalck, D. Jouan-Rimbaud, and D.L. Massart. The mahalanobis distance. *Chemometrics and Intelligent Laboratory Systems*, 50(1):1 – 18, 2000.
- [121] E. A. Stuart. Matching methods for causal inference: A review and a look forward. *Statistical Science*, 25:1 – 21, 2010.
- [122] Vijay D’Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008.
- [123] Stephen G. Brush. History of the lenz-ising model. *Rev. Mod. Phys.*, 39:883–893, Oct 1967.
- [124] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.
- [125] L.A. Zadeh, G.J. Klir, and B. Yuan. *Fuzzy sets, fuzzy logic, and fuzzy systems: selected papers*. Advances in fuzzy systems. World Scientific, 1996.
- [126] Vilém Novák, I. Perfilieva, and J. Močkoř. *Mathematical principles of fuzzy logic*. The Kluwer international series in engineering and computer science. Kluwer Academic, 1999.
- [127] Lotfi A. Zadeh. A summary and update of fuzzy logic. In *IEEE International Conference on Granular Computing*, volume 0, pages 42–44, Los Alamitos, CA, USA, 2010. IEEE Computer Society.
- [128] Ross Gore, David Kamensky, and Paul F. Reynolds Jr. Applying enhanced fault localization technology to monte carlo simulations. In *Proceedings of the 43rd Conference on Winter Simulation, WSC ’11*, pages 2798–2809, San Diego, CA, USA, 2011. Society for Computer Simulation International.
- [129] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, March 2001.
- [130] Zhenyu Zhang, W.K. Chan, and T.H. Tse. Fault localization based only on failed runs. *IEEE Computer*, 45(6), 2012.
- [131] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.
- [132] Esp source code repository. <https://riouxsvn.com/svn/esp/>.