APPLICATIONS OF STATIC ANALYSIS AND PROGRAM STRUCTURE IN STATISTICAL DEBUGGING

by

Piramanayagam Arumuga Nainar

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2012

Date of final oral examination: 8/17/2012

The dissertation is approved by the following members of the Final Oral Committee:

Benjamin R. Liblit, Associate Professor, Computer Sciences

Susan B. Horwitz, Professor, Computer Sciences

Shan Lu, Assistant Professor, Computer Sciences

Thomas W. Reps, Professor, Computer Sciences

Xiaojin Zhu, Associate Professor, Electrical and Computer Engineering

Copyright © 2012 Piramanayagam Arumuga Nainar

All Rights Reserved

To *amma* and *appa*

Acknowledgments

I thank my advisor Ben Liblit for his guidance and constant support. He has been a great mentor and guide both academically and personally. Ben has given me a lot of freedom to explore my interests, while still steering me towards achievable goals. His patience with me, especially during periods of fruitless ideas, has been unimaginable.

I would like to thank professors Susan Horwitz, Shan Lu, Tom Reps, and Jerry Zhu for their valuable feedback and comments as a part of my final oral committee. Comments from Susan helped bring a better structure and clarity to the chapter on identifying failure-inducing changes.

I am grateful to Anne Mulhern, Cindy Rubio González, Steve Jackson, Tristan Ravitch, Suhail Shergill, Mark Chapman, and Peter Ohmann for the discussions and their feedback during our group meetings. I am thankful to Jake Rosin and Ting Chen for their collaboration in the Compound Predicates project. Special thanks to Gogul Balakrishnan, Akash Lal, Nick Kidd, Junghee Lim, Evan Driscoll, Aditya Thakur, Matt Elder, Bill Harris, Tushar Sharma, Prathmesh Prabhu, and Venkatesh Srinivasan for their feedback, especially during my practice talks.

Several research groups supported this research by generously providing us with their tools and support. Our binary instrumentor is built using Dyninst. Prof. Bart Miller, Drew Bernat, and Matt Legendre, helpfully responded to our queries, issues, and feature requests regarding Dyninst. Grammatech generously provided us with licenses to CodeSurfer, which is used extensively by this research. The UWCS Condor pool, maintained by the Condor project, has been very useful for conducting the numerous experiments in this dissertation. Cathrin Weiss helped set up performance measurements on Condor. I would like to thank all the members of Dyninst, Grammatech, and Condor for their help and support.

I am thankful to all my friends for making my stay at Madison enjoyable, and the winters bearable. Special thanks to Arunachalam, Aditya, Asim, Cindy, Gautam, Giri, Jayaram, Jayashree, Keerthi, Leo, Nilay, Rathijit, Sayani, Sridhar, Srinath, Sriram, Swami, Tushar Khot, and Tushar Sharma for their friendship over the years. Sriraam, I enjoyed all the hours we spent following cricket, as well as our impromptu road trips. Ashok, you are all one can ask for in a roommate — a great friend, understanding roommate, and a great cook. Siddharth, I cherish our friendship that is forged in our mutual love for coffee, tennis, music, biking, and "humor".

I dedicate this dissertation to my parents for their love and affection, and for inspiring me during every step of my life. Karthi and Paru, our journey together over the years has been wonderful. Thank you!

Contents

Contents iv

List of Tables viii

List of Figures x

Abstract xii

1 Introduction 1

1.1 *Cooperative Bug Isolation* 1

1.1.1 Relevance of Static Program Information 4

1.2 *Offline Analysis of Compound Predicates* 4

1.3 *Adaptive Instrumentation* 5

1.4 *Identifying Failure-inducing Changes* 7

1.5 *Contributions* 8

2 Background 10

2.1 *Instrumentation Schemes* 10

2.2 *Sampling* 12

2.3 *Statistical Bug Isolation* 13

2.3.1 Scoring Predicates 14

2.3.2 Iterative Bug Isolation 15

2.4 Evaluation of the Output of Fault Localization 15

3 Compound Predicates 17

3.1 Compound Predicates: Formalization 18

3.1.1 Deriving Compound Predicates 19

3.1.2 Pruning Computation of Scores 21

3.2 Usability Metric 30

3.3 Case Studies 32

3.3.1 `exif` 32

3.3.2 `print_tokens` 35

3.4 Experiments 36

3.4.1 Top-scoring Predicates 37

3.4.2 Bug-relevance of Compound Predicates 38

3.4.3 Experiments on Larger Benchmarks 41

3.4.4 Effectiveness of Pruning 42

3.4.5 Effect of Effort and Sampling 44

3.5 Summary 48

4 Adaptive Bug Isolation 50

4.1 Overview of Adaptive Bug Isolation 52

4.1.1 Practical Considerations 54

4.2 Binary Instrumentation 57

4.2.1 Basic Instrumentation and Reporting 57

4.2.2 Static Removal of Instrumentation 58

4.2.3 Binarization and Dynamic Removal 59

4.2.4 Performance Impact 60

4.3 Adaptive Instrumentation 61

4.3.1 Forward Analysis of the Program 63

4.3.2 Backward Analysis of the Program . . . 64
4.3.3 Scoring Heuristics . . . 65
4.3.4 Waiting for Sufficient Data . . . 68
4.3.5 Design Alternatives . . . 69
4.4 Evaluation . . . 69
4.4.1 Comparison of Heuristics . . . 71
4.4.2 Instrumentation Selectivity . . . 75
4.4.3 Multiple Bugs . . . 78
4.4.4 Performance Impact . . . 80
4.4.5 Comparison with Holmes . . . 81
4.5 Summary . . . 82

5 Identifying Failure-inducing Changes 84
5.1 Conditional Coverage Estimation . . . 86
5.1.1 Basic Definitions . . . 87
5.1.2 Estimating Co-occurrence of Nodes . . . 89
5.1.3 Handling Function Calls . . . 92
5.1.4 Recursion and Loops . . . 93
5.1.5 Sources of Imprecision . . . 96
5.1.6 Change Impact Analysis . . . 96
5.2 Experimental Setup . . . 98
5.2.1 Performance . . . 99
5.3 Isolating Failure-inducing Changes . . . 100
5.3.1 Associating Changes with Predictors . . . 101
5.3.2 Associating Changes with Failing Locations . . . 103
5.3.3 Association using Forward Slices . . . 104
5.4 Summary . . . 105

6 Related Work 107

6.1 Survey of Statistical Debugging Techniques 107

6.1.1 Extensions from the CBI Project 107

6.1.2 Fault Localization Tools 108

6.2 Related Work for Compound Predicates 110

6.3 Related Work on Adaptive Instrumentation 111

6.4 Related Work for Conditional Coverage Profiles 113

6.4.1 Change Impact Analyses 113

6.4.2 Identifying Failure-inducing Changes 114

6.4.3 Probabilistic Static Analyses 115

7 Conclusion 116

7.1 Interoperability of the Contributions 117

7.2 Closing Thoughts 118

Bibliography 119

List of Tables

1.1 Relative performance overheads . . . 6

3.1 Three-valued truth tables for compound predicates . . . 21
3.2 Bounds required in equation (3.4) for a conjunction . . . 29
3.3 Bounds required in equation (3.4) for a disjunction . . . 29
3.4 Results for `exif` with only simple predicates . . . 33
3.5 Results for `exif` with compound predicates . . . 34
3.6 Results for `print_tokens` with simple predicates . . . 35
3.7 Results for `print_tokens` with compound predicates . . . 35
3.8 Properties of applications in the Siemens test suite . . . 36
3.9 Kind of the top predicate during complete data collection . . . 37
3.10 Kind of the top predicate during $1/100$ sampling . . . 38
3.11 Bug-relevance of simple and compound predicates . . . 39
3.12 Bug-relevance of top-ranked compound predicates found with and without *effort* metric . . . 40
3.13 Properties of larger bug benchmarks . . . 41
3.14 Kind of the top predicate for large bug benchmarks . . . 41
3.15 Time (in minutes) taken to compute scores for compound predicates . . . 42

4.1 Taxonomy of changes, with the proposed system being the bottom right cell . . . 55
4.2 Programs used for experimental evaluation . . . 70
4.3 Mean number of sites instrumented per iteration . . . 77

4.4 Mean number of iterations for each program 77

4.5 Results for `exif` 79

4.6 Relative performance overheads 80

5.1 Bug benchmarks used in experiments 98

5.2 Overhead of data collection and time for analysis 99

List of Figures

1.1 Overview of the CBI system . . . 3
1.2 Summary of contributions . . . 9

3.1 Illustration for $\uparrow R(C \text{ obs})$ . . . 24
3.2 Illustration for $\downarrow R(C \text{ obs})$ . . . 25
3.3 Illustration for $\uparrow R(D \text{ obs})$ . . . 27
3.4 Illustration for $\downarrow R(D \text{ obs})$ . . . 28
3.5 Code snippet from function `exif_mnote_data_canon_load` in `exif` . . . 33
3.6 Code snippet from function `get_token` in `print_tokens` . . . 35
3.7 Percentage of predicates pruned using *effort* and upper-bound in *Importance* . . . 43
3.8 Variation in the number of interesting predicates with *effort* . . . 45
3.9 Sampling rate vs. number of interesting predicates . . . 46
3.9 Sampling rate vs. number of interesting predicates (cont.) . . . 47

4.1 Module structure in `exif 0.6.9` . . . 51
4.2 Overview of adaptive bug isolation (a dark shadow highlights differences from fig. 1.1) . . . 53
4.3 Example graphs for static removal of branch predicates . . . 58
4.4 Adaptation speed for various heuristics using forward analysis . . . 72
4.5 Adaptation speed for various programs using forward and backward analysis . . . 74
4.6 Mean number of sites to find top-ranked predictor . . . 76

5.1 Example control-flow graph fragment 91
5.2 Control-flow graph for a simple square root computation 94
5.3 General equations for conditional coverage profile for the CFG in fig. 5.2 95
5.4 Conditional coverage profile from node n_1 for the CFG in fig. 5.2 95
5.5 Conditional coverage profile from node n_{11} for the CFG in fig. 5.2 95
5.6 Example illustrating change impact analysis 97
5.7 Precision and recall of associating failure-inducing changes with bug predictors 102
5.8 Precision and recall of associating failure-inducing changes with failing locations 103

Abstract

Software testing is costly, time-consuming, and often incomplete. Statistical debugging is a new domain of research that extends the testing phase into deployment. Post-deployment monitoring and statistical analyses are used to find program behaviors, called *predicates*, that are correlated with failures. Prior works rely on dynamic analysis, which focuses solely on the runtime behavior of programs, for bug isolation. This dissertation demonstrates the use of static analysis, a counterpart of dynamic analysis, to improve statistical debugging. Static analysis discovers a program properties without running it.

The contributions are evaluated in the context of the Cooperative Bug Isolation (CBI) project that studies statistical debugging techniques. Predicates instrumented by CBI test the program's state at particular program locations. At a high level, predicates are considered useful for fault localization if the set of executions in which they are observed true closely matches the set of failed executions. However, complex bugs manifest only when multiple factors co-occur. Simple predicates may not be accurate predictors of complex bugs. We propose that compound predicates, which are propositional combinations of simple predicates, could be better failure predictors. Compound predicates are the most accurate predicates of failure in 93% of our experiments. We use offline estimation, mathematical upper-bounds on failure predictivity, and static program dependences to tractably incorporate compound predicates into statistical debugging. These optimizations reduce analysis time from 20 minutes to just 1 minute.

Statistical debugging searches for needles in a haystack: over 99.996% of predicates are not failure predictive. The CPU, network, and storage resources used to collect these predicates could be better utilized. We develop an adaptive bug isolation technique that uses static program dependences and prior feedback to selectively monitor those predicates that are likely to be useful. Predicates that are irrelevant

to failure are never instrumented, or are removed after their relevance is ascertained. We characterize this adaptive predicate selection as a forward analysis on the program-dependence graph. We also develop a backward analysis that uses a crash location as a starting point. Our approach finds the best predicate found by complete instrumentation after exploring, on average, just 40% of the program locations. More importantly, very few locations are instrumented at any time, yielding imperceptibly low overheads of less than 1%.

While debugging, a developer first uses the list of predicates in CBI's output to find root causes of failures, and then develops a fix. We aid the developer in the first task by augmenting CBI's output with source-level changes that are likely causes of failures. We extend the well-studied problem of change impact analysis to compute the likelihood that a source-level change impacts a program location. Our technique, called conditional coverage profiles, uses symbolic evaluation, the program's control-flow graph, and runtime profiles to narrow the scope of impacted program locations by an order of magnitude compared to prior work. It identifies failure-inducing changes with a precision of 89% and a recall of 55%.

Our contributions, while seemingly orthogonal in their goals, are unified by their application of static analysis to connect dynamic behavior. Static program structure is used to prune less useful compound predicates, adaptively instrument likely bug predictors, and associate predicates with failure-inducing changes. Overall, this dissertation improves the monitoring efficiency and fault localization of the state-of-the-art in statistical debugging.

Chapter 1

Introduction

Statistical debugging tools analyse execution profiles collected at runtime to find bugs. Static analysis and program-dependence graphs (PDGs) can be used to improve the fault localization ability and monitoring efficiency of statistical debugging techniques.

Statistical debugging via post-deployment monitoring is a new domain of research that extends the testing phase into deployment. Such tools record only a small fraction of a program's runtime behavior and rely on the economy of scale and statistical analysis for effective fault localization. The design of these techniques have twin goals of

(a) gathering meaningful data for effective fault localization, and

(b) reducing the overhead of monitoring to facilitate post-deployment data collection.

Both these constraints must be met for the widespread adaptation of post-deployment monitoring. Static program structure and static analysis can help us achieve this goal.

1.1 Cooperative Bug Isolation

Software has bugs. Software testing is one method of ensuring software quality and reliability. However, software testing is costly, time consuming, and often incomplete. Due to market and time constraints, software is often released with bugs. Regression testing, feature testing, and program verification are

some methods used by software engineers to test the correctness and reliability of software. However, it is infeasible, and even impossible to test a program on all possible inputs and configurations. Bugs in a deployed program can arise during the execution of an untested code path or deployment in an untested environment.

Fixing bugs that affect the actual users of the program, and hence affect their perception of the software quality, is a higher priority for developers than fixing known, but rare, bugs found via in-house testing. Currently, public bug trackers and automatic crash reporting are the conventional means by which the software developer keeps track of bugs that arise after deployment. Post-deployment monitoring [Libb] has been proposed as a more heavyweight alternative to crash reporting. Unlike crash reporting, which gets invoked only at the time of a failure, post-deployment monitoring is always enabled. Diagnostic information about a program's execution is continuously collected. After every program run, this diagnostic information gets automatically uploaded to a central server, and is available for offline analysis. Bug isolation techniques analyze this data to identify *bug predictors*. A bug predictor is a program behavior that is highly correlated with program failure. This information can then be used by a software developer to diagnose and fix the bug.

Post-deployment monitoring and the associated bug-isolation techniques are collectively referred to as *statistical debugging*. Cooperative Bug Isolation is a project that studies statistical debugging techniques. Figure 1.1 shows a conceptual overview of this project. A source-to-source instrumentor modifies the program to record runtime behavior. An *instrumentation site*, the basic unit of instrumentation, inspects the state of the running program at a single program location. For example, a *branch* site inspects the outcome of each conditional statement. A *scalar-pairs* site inspects the relation between the output of a scalar assignment statement and every other scalar value in scope. A *return* site inspects the relation between a scalar value returned by a function call and zero. The inspiration of the return instrumentation scheme is the convention used in C programs where the return value indicates the success or failure of an operation. The CBI instrumentor supports several other instrumentation schemes [Liba]. Each instrumentation site is decomposed into a small collection of *instrumentation predicates* that partition the state space at that site. For example, there are two predicates at a branch site: one testing whether the branch condition is true and

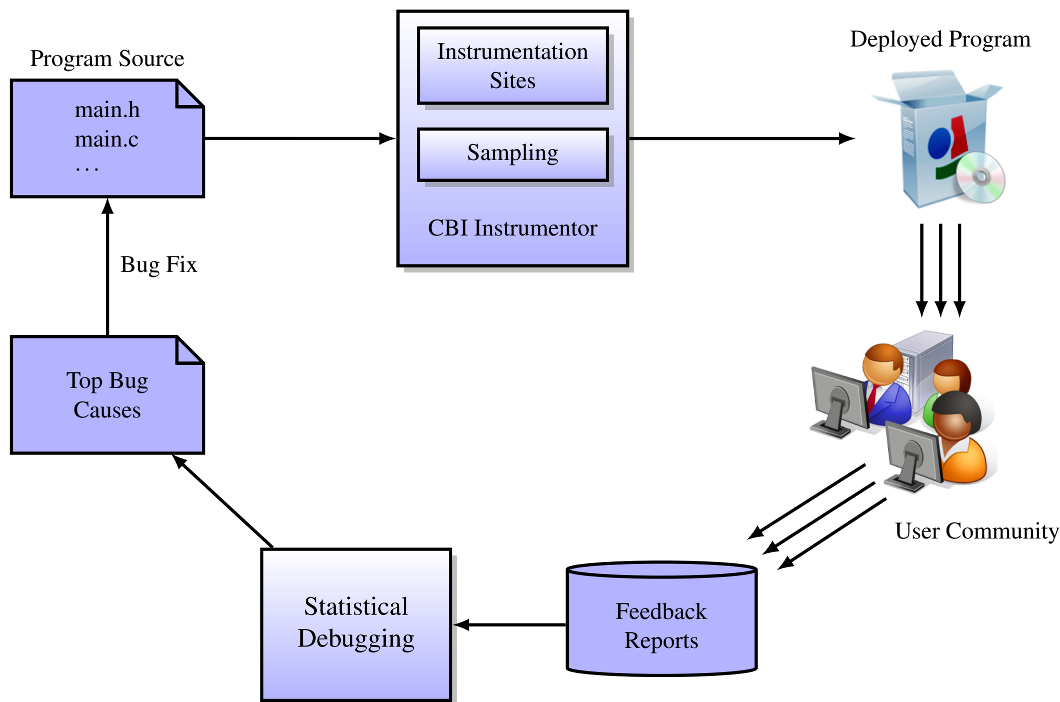


Figure 1.1: Overview of the CBI system

another testing whether the condition is false. These instrumentation schemes are a wide net intended to catch useful clues for a broad variety of bugs. They might be both incomplete and mutually redundant with respect to possible program behavior. CBI also uses sampling [Lib+03] to monitor a sparse, but fair random subset of the dynamic instances of each instrumentation site. At a sampling rate of $1/100$, every time the program reaches an instrumentation site, there is a one in a hundred chance that the predicates at that site are recorded.

After deployment, each run of the program generates a compact trace of the values of instrumentation predicates. The trace, and a single outcome label that marks the run as good (successful) or bad (failed) is uploaded to a central server. Fault isolation algorithms based on statistical techniques [Lib+03; Lib+05], and machine learning [And+07; Zhe+06] have been developed to analyze this data. In addition to success on widely used bug benchmarks, they also found previously unknown bugs [Lib+05]. The output of these techniques is a list of predicates whose truth in an execution of the program is correlated with failure. The programmer is presented with a list of such predicates, as well as the statistical reasoning behind this

decision. This information, hopefully, steers the programmer to the root cause of the bug and aides in the development of a fix. Over time, this continual process can improve the quality and reliability of the software.

1.1.1 Relevance of Static Program Information

The balance between accurate fault localization and efficient data collection is evident in the design of CBI. Scalar-pair instrumentation sites are defined at each assignment to a scalar variable, and hence can be quite numerous. Other instrumentation schemes like *g-object-unref* and *float-kinds* [Liba] can be included depending on the properties of the program being tested. These choices enhance the ability to find a broad variety of bugs. On the other hand, sampling throws away a vast majority of the data to guarantee minimal performance overhead. To reduce space consumption, the feedback reports are compacted to collect the frequency, rather than a complete trace, of predicate observations. Even though there are several instrumentation schemes, complex schemes such as path-profiling are eschewed in favor of simple properties.

Within the framework of post-deployment monitoring, there is a rich design space, and scope for new techniques for program instrumentation and statistical analyses. Static analysis and program dependences can be used to improve the state-of-the-art in this domain. The rest of this chapter makes various proposals that use this information to improve the results of fault localization and the overhead of data collection.

1.2 Offline Analysis of Compound Predicates

An instance of the trade-off between accurate fault localization and efficient data collection is the choice of predicates instrumented by CBI. The branch, return and scalar-pairs instrumentation schemes, as well as other instrumentation schemes supported by CBI are simple properties that can be quickly tested at runtime. However, such predicates might not be the best *predictors* of program failure. At a very high level, a predicate can be thought of as partitioning the space of all runs into two subspaces: those satisfying the predicate and those not satisfying it. A bug in the program can be considered as partitioning the space of runs into successful and failed runs. If the partitions induced by a predicate closely matches the set of

failed runs, , it would be considered as a good predictor for the bug. However, real-world bugs exhibit themselves in complex situations. The partition of executions induced by a predicate may only crudely match the set of failed executions. Such simple predicates may not always be the best predictor of the bug.

On the other hand, predicates added by complex instrumentation schemes can induce complex partitions of the set of executions. For example, a path predicate is true only in the intersection of the runs in which each branch in that path is true. Thus, path predicates *may* be better predictors of complex bugs. However, complex instrumentation schemes impose higher runtime overheads. A middle ground between simple instrumentation schemes and direct profiling of complex schemes is to consider Boolean formulae of simple predicates as candidate bug predictors. The predicates added by CBI's instrumentation schemes are Boolean conditions on the program state. A natural way to combine them into *compound* predicates is using logical operators such as conjunction and disjunction. We add compound predicates as candidate bug predictors during offline analysis. This approach imposes the same performance overhead as existing CBI instrumentation. The challenges in introducing compound predicates lie in the identification of compound predicates that are good failure predictors, as well as of better use to programmers in tracking the bug. We use statistical estimation and static program dependences to efficiently compute those compound predicates that are likely to be useful bug predictors.

1.3 Adaptive Instrumentation

CBI uses lightweight instrumentation based on sampling to reduce the runtime overhead of post-deployment monitoring. At sampling rates between $1/100$ and $1/1000$ suggested by Liblit [Lib07], more than 99% of the predicate behavior is never recorded. The compromise made by sampling is to uniformly discard more than 99% of the runtime profile. However, Liblit [Lib07] also found that more than 99% of the instrumented predicates are not predictive of failure. In such a situation, uniformly applying sparse sampling is not the optimal approach. Instead, we can combine intermediate fault localization results and program dependences to prioritize data collection towards those sites that are more likely to execute in a failed execution. Such sites are more likely to be useful in fault localization.

At a high level, such an adaptive predicate selection will work as follows. We start by monitoring an

Table 1.1: Relative performance overheads

	Sampling Rate		
Program	$1/1$	$1/100$	1/UINT_MAX
`bash`	1.315	1.257	1.137
`bc`	1.172	1.146	1.130
`gcc`	3.686	2.426	1.651
`gzip`	3.583	2.012	1.565
`exif`	1.893	1.975	1.292
Overall	2.076	1.692	1.338

initial set of instrumentation sites. Feedback obtained during this stage can be used to choose sites that *could* be causing failures. This set of sites is monitored during the next stage. More importantly, those sites that cannot be correlated with failure can be identified and their instrumentation can be postponed until a later time. Throughout this process, statistical-analysis results are available to the programmer, who can fix failures if enough data is available or choose to wait for more data if the picture is unclear. Intuitively, such an adaptive predicate selection tries to mimic the debugging activity of a programmer. Using feedback from a prior execution, or even just an initial hunch, the programmer uses breakpoints and other probes near points of failure to get more feedback about program behavior. Suspect code is examined more closely, while irrelevant code is quickly identified and ignored. Each iteration enriches the programmer's understanding until the reasons for failure are revealed. We propose to mimic and automate this process on a large scale. Instead of a single run, we can collect feedback from thousands or millions of executions of the program by its users. The adaptive step uses the control-dependence graph of the program to choose the set of predicates that get instrumented in the next iteration.

An added motivation for studying adaptive re-instrumentation techniques is that sampling does not improve performance in all cases [Lib07]. In particular, the overhead is non-trivial for CPU intensive workloads. Table 1.1 shows the average overhead of sampling-based instrumentation for some CPU intensive bug benchmarks. The overheads are obtained by normalizing against the running time of the uninstrumented program. Sampling rates of 1, $1/100$ and 1/UINT_MAX are studied. At a sampling rate of 1/UINT_MAX, virtually no feedback gets collected. It is the best case for sampling based instrumentation. Even for

this scenario, the overheads range from 1.29× to 1.65×, with a mean overhead of 1.338×. The adaptive instrumentation technique proposed above will also reduce the performance overheads because at any point in time, only a small subset of the instrumentation sites will be instrumented. Moreover, by selecting a small set of sites to instrument at any point in time, we can observe the behaviors of these sites in their entirety. Sampling is no longer required to guarantee low overheads.

1.4 Identifying Failure-inducing Changes

The output of statistical debugging tools is a list of predicates that are good predictors of failure. The programmer who uses these tools has two tasks at hand: first using failure predictors to identify the root cause of program failures, and then developing a fix for it. We can augment this list to also direct the attention of the programmer to possible root causes. In the context of evolutionary software development, the notion of a "root cause" could be a change made to the program from an earlier version. For example, if the new version of a program has a bug that was not present in an old version, the set of changes made to the old version may be the direct or indirect cause of the bug. Such changes are likely root causes for bugs introduced during minor releases of the software. For major releases, changes can build entire new features. Hence isolating a single change in a major release would be less useful in identifying a smoking gun. Nevertheless, identifying failure-inducing changes is a useful enhancement to the output of statistical debugging tools.

This proposal is inspired by other tools that identify failure-inducing changes. Given a passing and failing version of a program and the set of atomic changes made between the two versions, the goal of these dynamic-analysis tools is to find the minimum set of changes that cause failure when applied. However, existing work assumes the ability to enable each change individually [RR07; Zel99; Zha+08] or access to precise coverage information [Hof+09; Stö+06]. Such precise coverage information is not available in the presence of sampling or adaptive predicate selection. Thus, they are best suited for in-house testing, where it is possible to discover failures and collect profiles with a subset of changes enabled.

The problem of identifying failure-inducing changes can be solved by drawing on another rich body of research. The well-studied problem of Change Impact Analysis [Arn96] identifies the components

that are potentially impacted by a change to a program. If we find the source-level change that impacts a failure predictor found by statistical analysis, we can classify that change as a potential root cause of this failure. However, the output of existing change impact analyses [Api+05; Bin97; Hor+90; Hut+94; Kun+94; Ren+04; SH10; Yin+04] is ill-suited for this task. The output of these analyses is a Boolean decision on whether a particular program point is impacted. From our experience, a failure predictor is likely to be impacted by multiple changes.

Instead, we can quantify the impact of the change on each program point as a more nuanced, probabilistic value. This value is derived from branch and call-graph profiles of the old and new versions of the program. Our technique solves a data-flow problem similar to prior work on "Data flow frequency analysis" [Ram96]. The data-flow problem takes into account the location of the source-code change. We can use this probabilistic value to identify the change that is most likely to impact a failure predictor. We can objectively evaluate the accuracy of this classification. However, it is hard to evaluate whether information about a failure-inducing change is more useful to the programmer than just a failure predictor. Subjectively, the failure-inducing change, if correct, is a better starting place for a programmer to attempt a fix.

1.5 Contributions

Figure 1.2 summarizes our contributions along two criteria: improving failure prediction and efficiency of monitoring. The idea of compound predicates, further developed in chapter 3, adds Boolean combinations of predicates as candidate bug predictors. Chapter 5 explores the idea of associating a failure-inducing change with each failure predictor. Both of these proposals aid the programmer's fault comprehension, while using the same data as current techniques. Adaptive predicate selection, explained in chapter 4, attempts to find the same failure predictors as current techniques while significantly reducing the overhead of data collection. The proposal of identifying failure-inducing changes is orthogonal to adaptive predicate selection. Theoretically, the failure predictors and edge profiles collected using adaptive predicate selection can be used to find failure-inducing changes. On the other hand, compound predicates are derived using offline aggregation of the per-run predicate information. By limiting the set of predicates monitored in any run, adaptive predicate selection limits the set of compound predicates that can be used for bug isolation.

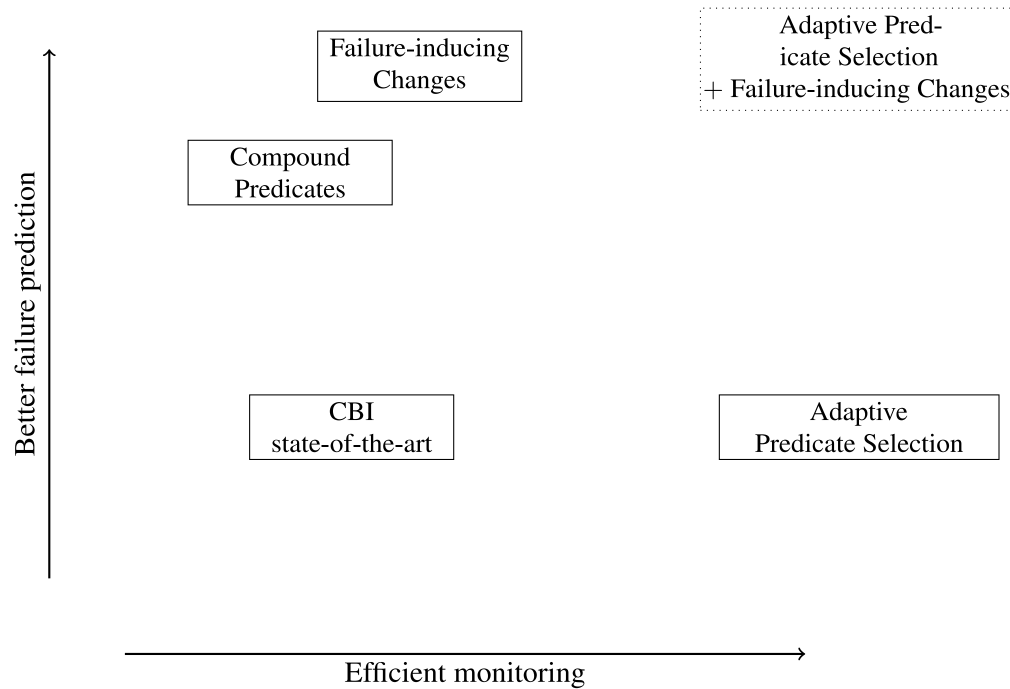


Figure 1.2: Summary of contributions

Research on post-deployment monitoring must take two kinds of users into account. First are the programmers who analyze the output of statistical analyses to understand and fix a bug. Compound predicates and failure-inducing changes are qualitative extensions to the output of statistical analyses that help the programmer in his task. The second category of users are the actual users of the software. They are indirect beneficiaries of post-deployment monitoring. They also form the foundation of post-deployment monitoring by allowing collection of diagnostic information. Adaptive predicate selection helps them by reducing the overhead of data collection.

Chapter 2

Background

This chapter introduces background information about CBI to aid the discussion in the following chapters. It explains the instrumentation [Lib+03] and analysis [Lib+05] phases of CBI in detail. Chapter 6 surveys other work in the field of statistical debugging, and explains the relevance of the ideas in this dissertation to the related work.

2.1 Instrumentation Schemes

The basic unit of instrumentation added by CBI is called an *instrumentation site*. An instrumentation site is defined at a single program point where the state of the running program is inspected. Each site is decomposed into a small collection of *predicates*, each of which corresponds to a Boolean condition on the state of the program. The predicates partition the state space at that site. Instrumentation sites are automatically selected based on the syntactic features of the code. They may be both incomplete and mutually redundant with respect to possible program behavior; they are not a perfect execution trace, but rather are a wide net intended to catch useful clues for a broad variety of bugs.

The CBI instrumentor supports several schemes of instrumentation sites, each associated with a particular syntactic feature of the program. However, prior work has found the branch, return and scalar-pairs schemes to be the most useful for purposes of fault localization. This section explains these three schemes in detail. We also limit instrumentation to these three schemes for experiments in subsequent

chapters.

- A *branch* site is defined at each conditional statement and tests the outcome of the condition. The predicates at a branch site split the state space into two sets: one where the branch condition is true and another where it is false.
- A *return* site is defined at each function call that returns a scalar value. The predicates at a return site split the state space into three sets, where the return value is negative, zero, and positive. This partition is especially well-matched to C programs, as the sign of a returned value often indicates success or failure of an operation.
- At an assignment to a scalar variable x in the program, multiple *scalar-pairs* sites are instrumented, one for every scalar variable y in scope. The predicates at a scalar-pair site split the state space into three sets, where $x < y$, $x = y$ and $x > y$.

Every time the execution reaches an instrumentation site, exactly one of the predicates at that site is true. For reasons of privacy, and to limit the size of the feedback reports, this stream of predicate observations is compressed into a vector of predicate counters. The counter corresponding to a predicate captures the number of times that predicate was true in that execution. The sum of all predicate counters at a site gives the overall coverage of that site. The feedback report for a run also has a single outcome label indicating whether this run was good (successful) or bad (failed). In the simplest case, failure can be defined as crashing, and success as not crashing. More refined labeling strategies are easily accommodated, as subsequent analysis stages do not care how the success/failure distinction was made. In particular, failure analysis does not use stack traces, and therefore can be applied to non-crashing bugs.

For sites that have more than two associated predicates, the complement of a predicate can also be considered as a predicate. For example, there are three additional predicates at a return site that capture whether the return value is nonnegative, nonzero or nonpositive. At a scalar-pairs site that compares a scalar variable x with another scalar variable y, there are three additional predicates that capture whether $x \leq y$, $x \neq y$ and $x \geq y$. At a branch site that has only two predicates, the predicates are complements of each other. Hence, no additional predicates are added.

2.2 Sampling

The instrumentation sites mentioned in the previous section can be quite numerous. It varies from around 500 sites for simple 500 line programs in the Siemens bug benchmark [Hut+94] to over 145,000 for `Rhythmbox`, which has about 57KLOC [Lib+05]. The computational overhead of gathering the feedback reports can be quite high. Sampling techniques have been developed that reduce this overhead. Instead of complete predicate counts, only a sparse but fair random subset of the counts is collected. Liblit et al. [Lib+03] have developed a sampling scheme based on a static source-to-source transformation applied at compile time. The transformation creates two copies of each function: a *slow* path with instrumentation enabled and a *fast* path without any instrumentation. The two copies are connected using control-flow constraints that switch execution to the slow path to sample an instrumentation site, and switch to the fast path at other times. The intuition behind this transformation is that at a sampling rate of, say, 1⁄100, the program executes along the fast-path for approximately 99% of the time. While executing the fast path, only a minor overhead for bookkeeping is imposed. This technique is derived from that of Arnold and Ryder [AR01]. Liblit [Lib07] concludes that sampling rates between 1⁄100 and 1⁄1,000 are suitable for realistic deployments. They find that sampling improves performance in some but not all cases. A sampling rate of 1⁄100 is used by the instrumented versions of popular open source software packages in the public deployment maintained by CBI. The overhead for these programs is deemed acceptable by the volunteer users.

One consequence of the sampling transformation is the blow-up caused by the fast- and slow- path versions of every function. Sampling approximately doubles the static code size, thereby increasing costs for packaging or network distribution. Avoiding code doubling may be mandatory for practical, economic reasons:

> We did the math of going to a second DVD for [Windows] Vista. Basically a second DVD doubles the costs, because you not only need two pieces of media, you also need a slightly more expensive case. M. Fortin [For07]

The doubling of code size, and the inability of sampling to guarantee low overheads for all cases are part

of the motivation behind the techniques for adaptive predicate selection introduced in chapter 4. Another consequence of sampling is that data analysis must cope with the fact that 99% or more of requested data is missing. The statistical analysis described in the next section has been designed to be able to handle such incomplete data.

2.3 Statistical Bug Isolation

The goal of statistical debugging techniques is to find *bug predictors*: predicates that, when true, herald failure due to a specific bug. Bug predictors highlight areas of the code that are related to program failure and so provide information that is useful when correcting program faults. This section describes the statistical debugging technique developed by Liblit et al. [Lib+05] in detail. These concepts are the most relevant to the ideas built in later chapters. Other statistical debugging techniques are discussed in section 6.1.

The inputs for these algorithms are the feedback reports collected from a set of executions. The feedback report for a run has a vector of counters, one per predicate. The predicate might be one that is directly instrumented or one that is derived offline. The counter denotes the number of times the predicate was observed to be true in this run. The sum of the counters corresponding to the directly instrumented predicates gives the number of times the predicates at that site were inspected. Aggregating these values across the entire suite of runs, we can find the number of runs in which the predicate was ever true, and the number of runs in which the predicate was ever observed. Each run is also labeled as successful or failed. By including this distinction, we get four values for each predicate p:

- $S(p \text{ obs})$ and $F(p \text{ obs})$, respectively the number of successful and failed runs in which the value of p was evaluated.
- $S(p)$ and $F(p)$, respectively the number of successful and failed runs in which the value of p was evaluated and was found to be true.

2.3.1 Scoring Predicates

These aggregated counts are used to compute two scores of bug relevance.

Sensitivity: $F(p)$ captures whether the predicate is observed true in a large number of failed runs. This property is normalized to *NumF*, the total number of failed runs, to get a score in the range $[0,1]$. It is also normalized using the logarithm function to moderate the impact of very large numbers of failures.

$$Sensitivity(p) \equiv \frac{\log(F(p))}{\log(NumF)} \tag{2.1}$$

Specificity: $Increase(p)$ captures the increase in the likelihood of failure when p is true over simply reaching the line where p is defined. It is computed as follows:

$$Increase(p) \equiv \frac{F(p)}{S(p)+F(p)} - \frac{F(p \text{ obs})}{S(p \text{ obs})+F(p \text{ obs})} \tag{2.2}$$

A high sensitivity score for a predicate indicates that p accounts for many failed runs. A high specificity score indicates that the predicate does not mis-predict failure in many successful runs. In information retrieval terms, sensitivity corresponds to *recall* of failure prediction and specificity corresponds to *precision* of failure prediction. A single score, $Importance(p)$ which captures the failure predictivity of p is computed by balancing the sensitivity and specificity scores as follows:

$$Importance(p) \equiv \frac{2}{\frac{1}{Increase(p)} + \frac{1}{Sensitivity(p)}} \tag{2.3}$$

The predicates are ranked using the *Importance* score. However, predicates might be mutually redundant. If there are failures corresponding to multiple bugs, the list of predicates usually has a lot of predicates corresponding to the most prevalent bug near the top. The iterative elimination algorithm described next is designed to handle situations with multiple bugs.

2.3.2 Iterative Bug Isolation

The process of iterative bug isolation starts by ranking predicates using the *Importance* score. The predicate at the top of this list is assumed to correspond to the most important bug, though other bugs may remain. This top predictor is recorded, and then all feedback reports where it was true are removed from consideration. The intuition behind this step is that fixing the corresponding bug will change the behavior of runs in which the predictor originally appeared. The scores of predicates are recomputed, and the next best predictor among the remaining reports is then identified, recorded, and removed in the same manner. This iterative process terminates either when no undiagnosed failed runs remain, or when no more failure-predictive predicates can be found.

This process of iterative elimination maps each predictor to a set of program runs. Ideally each such set corresponds to the expression of a distinct bug; unfortunately this is not always the case. Due to the statistical nature of the analysis, along with incomplete feedback reports resulting from sparse sampling rates, a single bug could be predicted by several top-ranked predicates, and predictors for less prevalent bugs may not be found at all.

2.4 Evaluation of the Output of Fault Localization

The output of the iterative bug isolation algorithm, and other related work in this field, is a ranked list of program properties that are *potentially* useful for debugging. In the case of CBI, the ranked list has instrumentation predicates. Prior work has used a metric [RR03] that considers the programmer as performing an undirected breadth-first search of the program-dependence graph (PDG) from each entry in the ranked list. This metric has two problems. First, case studies in prior work [Lib+05] find that patterns in the list of predictors, such as testing the value of a common variable, or proximity in the source code are important while debugging. Our case study in section 3.3.2 also finds this to be true. Second, a recent user-study on the usability of fault localization tools [PO11] finds that during debugging programmers do not process each item in the output list independently. They look at several items in the list, and not necessarily in the order in which they are ranked.

In light of this, just independent breadth-first searches from items in the ranked list of predicates is not the right evaluation metric. A qualitative evaluation of the results, in addition to being subjective, is time consuming. We qualitatively evaluate two scenarios in section 3.3. Prior work Liblit et al. [Lib+05] has shown that the *Importance* score is useful for finding previously unknown bugs, as well as seeded bugs in benchmarks. For evaluations in chapters 3 and 4, we use high *Importance* scores as a *proxy* for a useful bug predictor.

Chapter 3

Compound Predicates

CBI gathers feedback reports by using valuable CPU cycles at end user machines. It is essential to make those cycles worthwhile by extracting every bit of useful information from them. The instrumentation predicates considered by CBI test Boolean conditions on the current state of the program. We propose *compound predicates* which are propositional combinations of instrumentation predicates. The reasoning is that compound predicates induce complex partitions of the set of executions. Such partitions *may* match the set of failed runs, and hence the compound predicate may be a better predictor of complex bugs. In this chapter, we use the term *simple predicate* to denote predicates at instrumentation sites and the term *compound predicate* to denote propositional combinations of simple predicates.

As an example, consider a hypothetical bug due a *null-pointer* error. Two conditions must co-occur for this bug to manifest: a pointer must be assigned `null`, and the `null` pointer must later be dereferenced. One such bug exists in `exif` 0.6.9, and was found by prior work [Lib+05]. However, the key predicate that was used to find the bug did not have a have a high score and hence was ranked much lower in the output. A non-persistent user might be discouraged by irrelevant predicates (*false positives*) in the output and give up on the task. This useful predicate captures one of the prerequisites of the bug, the assignment of `null` to a pointer. However, it captures only a necessary condition for failure. It is not a sufficient condition because the pointer may never be dereferenced. A statistical analysis performed using compound predicates finds a conjunction of two predicates as the best predictor. The first predicate in the conjunction was the simple

predicate that captures the executions where the `null` assignment occurs. The second predicate captures the executions where the dereference of the problematic pointer happens. Both the predicates captured conditions that were necessary but not sufficient for failure. The conjunction of the two predicates is a necessary and sufficient condition of failure, and hence is the perfect predictor. We present a detailed case-study of this bug along with code snippets and fault localization results in section 3.3.1.

Compound predicates can be incorporated into statistical debugging by either (a) changing the instrumentor to explicitly monitor each compound predicate at runtime, or (b) estimating the value of each compound predicate from the values of its components offline. The first approach will yield a precise value but needs significant modifications to existing infrastructure. Furthermore, as we show later in this chapter, even with rigorous restrictions on the set of compound predicates chosen, the number of candidate compound predicates is asymptotically quadratic in the number of simple predicates. The time and space overheads of monitoring compound predicates will be far from reasonable. The offline-estimation approach will be less precise but requires only few modifications to existing infrastructure, and none to the instrumentor. Therefore, we design and implement compound predicates using this approach. Section 3.1 gives a precise definition of compound predicates and discusses how they can be computed efficiently. Section 3.2 defines a metric to characterize the usefulness of compound predicates in the task of debugging. Section 3.3 discusses two case studies that demonstrate the usefulness of compound predicates. Section 3.4 presents the results of experiments conducted on a large suite of buggy test programs, including an assessment of the effect of sparse random sampling on compound predicates.

3.1 Compound Predicates: Formalization

The most general propositional combination of N simple predicates is $\phi(p_1, p_2, \ldots p_N)$, where $p_1, p_2, \ldots p_N$ are simple predicates and ϕ is a Boolean function of N variables. There are 2^{2^N} such functions [Wei06]. By design, the negation of every CBI predicate is also a predicate. Hence, the number of candidate compound predicates is slightly smaller than 2^{2^N}, but is still exponential in N.

To reduce complexity, we only consider functions of two predicates. There are 16 (2^{2^2}) such functions for each pair of simple predicates. We further restrict our focus to just conjunctions and disjunctions since

they are the simplest, and most easily understood by programmers. The revised definition of a compound predicate is $C = \phi(p_1, p_2)$ where $\phi \in \{\vee, \wedge\}$. Conjunction and disjunction are commutative, and the reflexive cases ($p_1 \wedge p_1$ and $p_1 \vee p_1$) are uninteresting. This reduces the number of compound predicates to just $\binom{N}{2} = \frac{N(N-1)}{2}$ binary conjunctions and an equal number of binary disjunctions.

3.1.1 Deriving Compound Predicates

To incorporate compound predicates into statistical debugging, we need to find if a compound predicate was observed in each run, and if so, whether it was true at least once. This information is defined for simple predicates as follows:

Definition 1. *For a simple predicate p and a run r, it can either be observed true at least once, observed and never true, or never observed during the run.*

- *$r(p)$ is true if p was observed true at least once during r.*
- *$r(p)$ is false if p was evaluated at least once, but was never true during r.*
- *$r(p)$ is unknown if p was never observed during the run.*
- *$r(p\ obs)$ is true if $r(p)$ is either true or false.*

A predicate p might be unobserved during a run r, and hence $r(p)$ is unknown, either because the execution did not reach the line where it was defined, or when the execution did reach the line , it was not observed because of sampling.

Similar to Definition 1, we can define the truth of a compound predicate in a run as follows.

Definition 2. *For a compound predicate $C = \phi(p_1, p_2)$, $r(C)$ is true iff at some point during the execution of the program, C was observed to be true.*

The difficulty with this notion of compound predicates is that C must be explicitly monitored during the program execution. For example, $r(p_1) =$ true and $r(p_2) =$ true does not imply that $p_1 \wedge p_2$ is ever true at a single program point. p_1 and p_2 may be true at different stages of execution but never true at

the same time. Furthermore, when p_1 and p_2 appear at different source locations, there may be no single point in time at which both are even well-defined and therefore simultaneously observable. In order to be able to estimate the value of C from its components, we adapt a less time-sensitive definition as follows:

Definition 3. *For a compound predicate* $C = \phi(p_1, p_2)$*,* $r(C) \equiv \phi(r(p_1), r(p_2))$*.*

In other words, we treat r as distributive over ϕ, effectively removing the requirement that p_1 and p_2 be observed simultaneously. This can lead to false positives, because $r(C)$ may be computed as true when C is actually false at all moments in time. False negatives, however, cannot arise[1]. The assumption of the distributive property may have either a positive or negative impact on the *Importance* score of C depending on whether the run r failed or succeeded. With this definition, simultaneous observation of the predicates is not required to determine whether a compound predicate is observed in a run: p_1 and p_2 might be defined in different locations, possibly in different functions. The compound predicates found useful in the case-studies in section 3.3 are defined in different program locations, where simultaneous observation is not possible, and yet, provide useful clues for understanding the failures. Moreover, adopting Definition 3 allows offline estimation of whether C was observed in a run.

Because $r(p_1)$ or $r(p_2)$ may be unobserved, three-valued logic is required to evaluate $\phi(r(p_1), r(p_2))$ in Definition 3. For the statistical analysis introduced in section 2.3, it is enough to consider whether $r(p)$ was true or ¬true (either false or unknown). When constructing compound predicates, however, we can use the sub-cases of ¬true to take advantage of the short-circuiting properties of logical operators.

Consider a compound predicate $p_1 \wedge p_2$. If either $r(p_1)$ or $r(p_2)$ was false in a run r, then $r(p_1 \wedge p_2) =$ false, since one false value disproves a conjunction. If both $r(p_1)$ and $r(p_2)$ were true, then $p_1 \wedge p_2$ was observed to be true. Otherwise, the value of $r(p_1 \wedge p_2)$ is unknown. This is shown using a three-valued truth table in table 3.1a. The symbol '?' is used to denote that the predicate was unobserved.

Similarly one true value proves a disjunction. Hence, $r(p_1 \vee p_2)$ is true if either of $r(p_1)$ or $r(p_2)$ was true. Also, $r(p_1 \vee p_2)$ is false if both $r(p_1)$ and $r(p_2)$ are false. The truth table for the predicate $p_1 \vee p_2$ is shown in table 3.1b.

[1] Sampling introduces false negatives, though, and affects both simple and compound predicates

Table 3.1: Three-valued truth tables for compound predicates

(a) Conjunction: $p_1 \wedge p_2$

$p_1 \diagdown p_2$	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

(b) Disjunction: $p_1 \vee p_2$

$p_1 \diagdown p_2$	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

3.1.2 Pruning Computation of Scores

Even using the revised definition for compound predicates, their number would be quadratic in the number of simple predicates. A large number of the compound predicates formed by this procedure are likely to be useless in the analysis of the program. Certainly a compound predicate that is less predictive of failure than one of its components is useless. The component (simple) predicate is a better predictor of failure, and so the compound predicate adds nothing to the analysis. We use the *Importance* score introduced in section 2.3.1 as the measure of failure predictivity.

Definition 4. *A compound predicate $C = \phi(p_1, p_2)$ is "interesting" iff its Importance score is strictly greater than the Importance score of both its component predictors. i.e.*

$$Importance(C) > Importance(p_1), \text{ and}$$
$$Importance(C) > Importance(p_2)$$

In the case where the compound predicate has the same score as a component simple predicate, the simpler one is preferable. Keeping only interesting combinations of predicates reduces the memory burden of storing them, and helps ensure the utility of a compound predicate that is presented to the user.

Constructing a compound predicate C from its components requires generating two bits of information for each program run: whether $r(C)$ is true and whether $r(C \text{ obs})$ is true. This is aggregated across the set of successful runs, S and failed runs F to obtain four quantities. For $R \in S, F$,

- $R(C)$ is the number of runs $r \in R$ for which $r(C)$ is true.
- $R(C \text{ obs})$ is the number of runs $r \in R$ for which $r(C \text{ obs})$ is true.

While using the iterative elimination algorithm (section 2.3), the *Importance* score for each compound predicate is defined using $S(C)$, $F(C)$, $S(C \text{ obs})$ and $F(C \text{ obs})$. Computing these values takes $O(|S+F|)$ time. As CBI is meant to analyze deployed software, program runs potentially number in the hundreds of thousands, if not millions. Even for small test datasets with around a thousand runs, this computation takes around twenty minutes in our experiments. However, our experiments found that most of these compound predicates are not interesting (i.e., they have low *Importance* scores) and are not presented to the programmer. In such cases, the time spent computing the exact scores has been wasted. In this section, we develop upper-bound estimates on the score of a compound predicate from the aggregate values $S(p)$, $F(p)$ etc. of its component simple predicates. If this estimate falls below the threshold required for the predicate to be presented to the programmer, the exact score is not computed. For the common case, this reduces the complexity of evaluating a predicate from $O(|S+F|)$ to $O(1)$.

The threshold used for pruning can be derived in two ways:

- Only interesting compound predicates, as formalized in Definition 4, are retained; for a compound predicate $\phi(p_1, p_2)$ the threshold would therefore be **max(**$Importance(p_1), Importance(p_2)$**)**.
- During iterative bug isolation, only the predicate with the highest score is retained during each iteration. The threshold for $Importance(C)$ is therefore the highest score yet seen (including those of simple predicates).

To simplify the discussion of the estimation of upper-bounds, we introduce some new terms and notations. If $R \in \{F, S\}$ is the set of program runs under consideration, $R(\overline{p})$ is the number of runs in which predicate p was observed at least once but never observed true. It is equal to $R(p \text{ obs}) - R(p)$. For some unknown quantity x, let $\uparrow x$ and $\downarrow x$ denote estimated upper and lower bounds on the exact value of x, respectively.

The *Importance* score of a predicate was described in section 2.3.1. The score of a compound predicate

C is as given below:

$$Sensitivity(C) \equiv \frac{\log(F(C))}{\log(NumF)} \tag{3.1}$$

$$Increase(C) \equiv \frac{F(C)}{S(C)+F(C)} - \frac{F(C\ \text{obs})}{S(C\ \text{obs})+F(C\ \text{obs})} \tag{3.2}$$

$$Importance(C) \equiv \frac{2}{\frac{1}{Increase(C)} + \frac{1}{Sensitivity(C)}} \tag{3.3}$$

where *NumF* is the total number of failed runs. The upper bound on the *Importance* score of a compound predicate can be computed by maximizing $Increase(C)$ and $Sensitivity(C)$ under constraints based on the Boolean operator. $Importance(C)$, being the harmonic mean of these two terms, will likewise be maximized. From equation (3.2), an increase in $F(C)$ or $S(C\ \text{obs})$ or a decrease in $S(C)$ or $F(C\ \text{obs})$ will increase the value of $Increase(C)$. So an upper bound on $Increase(C)$ is

$$\uparrow Increase(C) \equiv \frac{\uparrow F(C)}{\downarrow S(C) + \uparrow F(C)} - \frac{\downarrow F(C\ \text{obs})}{\uparrow S(C\ \text{obs}) + \downarrow F(C\ \text{obs})} \tag{3.4}$$

Since *NumF* is a constant, maximizing $Increase(C)$ also maximizes $Sensitivity(C)$.

Bounds for a Conjunction

So far, we have used the term C to denote a compound predicate. For notational convenience, henceforth we use C to denote a conjunction of two simple predicates and D to denote a disjunction.

Consider a conjunction $C = p_1 \wedge p_2$ and a set of runs R. We know that among the $|R|$ runs, p_1 was observed in $R(p_1) + R(\overline{p_1})$ runs, and it was true at least once in $R(p_1)$ of those runs. Similarly, p_2 was observed in $R(p_2) + R(\overline{p_2})$ runs, and it was true at least once in $R(p_2)$ of those runs. Under these constraints, we need to fix the co-observations of p_1 and p_2 in such a way that the desired quantity is maximized or minimized.

From table 3.1a, C is observed true in a run if both p_1 and p_2 were observed true in that run. Equivalently, the set of runs in which C was observed true is the intersection of (a) the set of runs in which p_1 was observed true, and (b) the set of runs in which p_2 was observed true. The size of this intersection, and

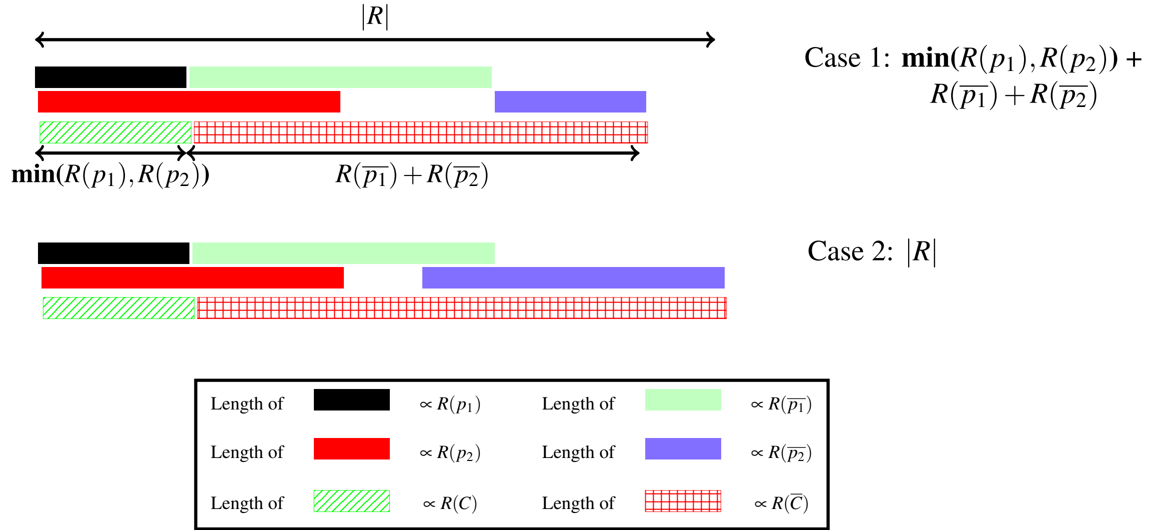


Figure 3.1: Illustration for $\uparrow R(C \text{ obs})$

consequently $R(C)$, cannot be larger than the either of these sets. Thus

$$\uparrow R(C) = \mathbf{min}(R(p_1), R(p_2))$$

Likewise, $R(C)$ is minimized when there is minimum overlap between the set of runs in which p_1 was observed true and p_2 was observed true. The minimum overlap is 0 as long as $R(p_1) + R(p_2)$ does not exceed $|R|$. Otherwise, both p_1 and p_2 must be observed together in at least $R(p_1) + R(p_2) - |R|$ runs. Thus,

$$\downarrow R(C) = \mathbf{max}(0, R(p_1) + R(p_2) - |R|)$$

Next, consider $R(C \text{ obs})$. Figure 3.1 illustrates the cases when $R(C \text{ obs})$ is maximized. The figure should be interpreted as follows. The line on the top is proportional to $|R|$, the size of the runs under consideration. Per the legend, each bar is proportional to the size of the set of runs in which the predicates under consideration, p_1, p_2, and C are observed true and observed but never true. For simplicity, we refer to the latter case as the predicates being observed false. As per table 3.1a, C is observed in the following cases.

Rule 1: If both p_1 and p_2 are observed true, C is observed true. In fig. 3.1, the bar proportional to $R(C)$ is

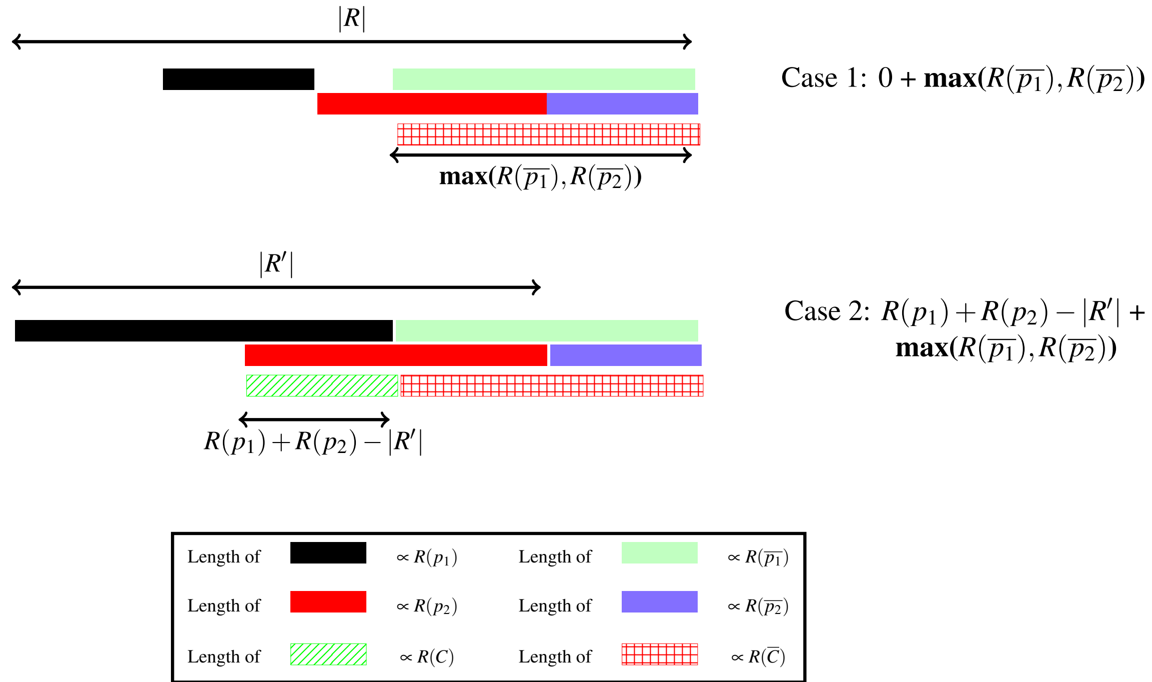


Figure 3.2: Illustration for $\downarrow R(C \text{ obs})$

active only when the bars denoting $R(p_1)$ and $R(p_2)$ overlap.

Rule 2: If either p_1 or p_2 is observed false, C is observed false. In fig. 3.1, the bar proportional to $R(\overline{C})$ is active when either of the bars proportional $R(\overline{p_1})$ and $R(\overline{p_2})$ are active.

To maximize $R(C \text{ obs})$, applications of both of these rules must be maximized. Applications of Rule 1 are maximized when the sets of runs in which p_1 and p_2 are observed true completely overlap. As shown in the illustration, C is observed true in $\mathbf{min}(R(p_1), R(p_2))$ runs in this case. Rule 2 is applied when either p_1 is observed false, or p_2 is observed false. Applications of Rule 2 are maximized when the sets of runs in which p_1 and p_2 are observed false are non-overlapping. In this case, C is observed false in $R(\overline{p_1}) + R(\overline{p_2})$ runs. In total, C is observed true in $\mathbf{min}(R(p_1), R(p_2)) + R(\overline{p_1}) + R(\overline{p_2})$ runs. However, there has to be an overlap between these cases if the sum exceeds $|R|$, the total number of runs. This is shown in Case 2 of fig. 3.1. In this case, C will be observed in all of the runs.

$$\uparrow R(C \text{ obs}) = \mathbf{min}(|R|, R(\overline{p_1}) + R(\overline{p_2}) + \mathbf{min}(R(p_1), R(p_2)))$$

Finally, to minimize $R(C \text{ obs})$, applications of the two rules mentioned above are minimized. Per Rule 2, the false observations of p_1 and p_2 must completely overlap to minimize the number of runs in which C is observed false. As shown in the illustration in fig. 3.2, C is observed false in $\mathbf{max(}R(\overline{p_1}), R(\overline{p_2})\mathbf{)}$ runs. This decides that neither p_1 nor p_2 was observed true in $\mathbf{min(}R(\overline{p_1}), R(\overline{p_2})\mathbf{)}$ of the runs. This leaves $|R'| = |R| - \mathbf{min(}R(\overline{p_1}), R(\overline{p_2})\mathbf{)}$ runs to pick, without overlap, the runs in which p_1 and p_2 are observed true. Such a non-overlapping arrangement will minimize the runs in which p_1 and p_2 are simultaneously observed true. If this is possible, applications of Rule 1 are minimized.

1. Suppose the sum $R(p_1) + R(p_2)$ is less than $|R'|$. As shown in Case 1 of fig. 3.2, the true observations of p_1 and p_2 can be made non-overlapping. Consequently, there are no runs in which the conjunction is observed true.

2. If $R(p_1) + R(p_2)$ exceeds $|R'|$, there will be a minimum overlap of $R(p_1) + R(p_2) - |R'|$ between the runs in which p_1 and p_2 are simultaneously observed true. As shown in Case 2 of fig. 3.2, the conjunction will be observed true in such runs.

Thus,

$$\downarrow R(C \text{ obs}) = \mathbf{max(}R(\overline{p_1}), R(\overline{p_2})\mathbf{)} + \mathbf{max(}0, R(p_1) + R(p_2) - |R'|\mathbf{)}$$

Bounds for a Disjunction

Consider a disjunction $D = p_1 \vee p_2$ and a set of runs R. From table 3.1b, D is observed true in a run if either p_1 or p_2 was observed true in that run.

Equivalently, the set of runs in which D was observed true is the union of (a) the set of runs in which p_1 was observed true, and (b) the set of runs in which p_2 was observed true. The size of this union, and consequently $R(D)$, will be at least as large as either of these two sets. Hence,

$$\downarrow R(D) = \mathbf{max(}R(p_1), R(p_2)\mathbf{)}$$

Likewise, $R(D)$ is maximized when there is no overlap between the set of runs in which p_1 and p_2 are observed true. If there is no overlap, $R(D)$ is equal to the sum of $R(p_1)$ and $R(p_2)$. If this sum exceeds $|R|$,

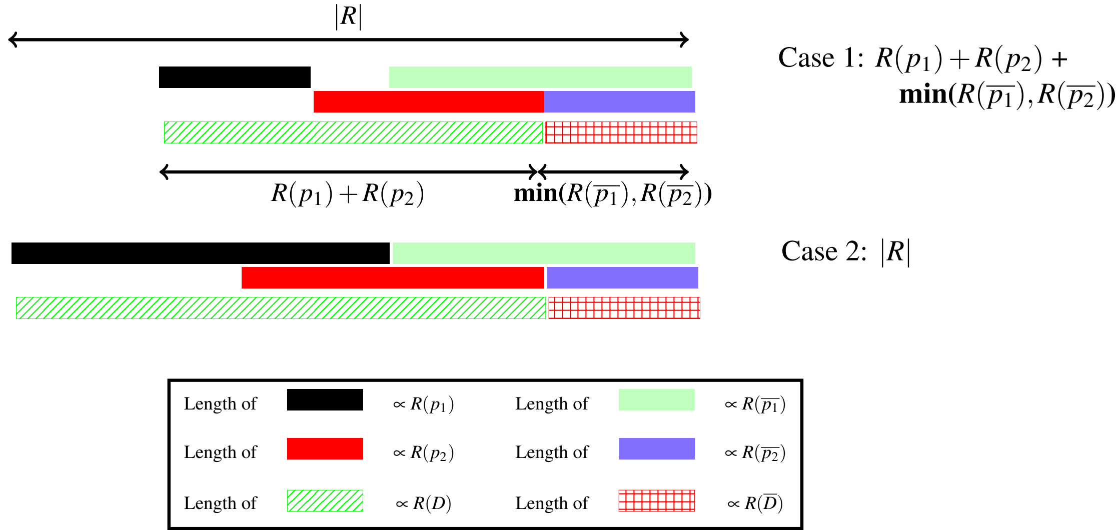


Figure 3.3: Illustration for $\uparrow R(D\text{ obs})$

the union of the two sets can, at best, be as large as R.

$$\uparrow R(D) = \mathbf{min}(|R|, R(p_1) + R(p_2))$$

Next, consider $R(D\text{ obs})$. Figure 3.3 illustrates the cases when $R(D\text{ obs})$ is maximized. D is observed in a run when

Rule 1: Either p_1 or p_2 is observed true, in which case D is observed true. In fig. 3.3, the bar proportional to $R(D)$ is active when either of the bars proportional to $R(p_1)$ and $R(p_2)$ are active.

Rule 2: Both p_1 and p_2 are observed false, in which case D is observed false. In fig. 3.3, the bar proportional to $R(\overline{D})$ is active only when the bars denoting $R(\overline{p_1})$ and $R(\overline{p_2})$ overlap.

To maximize $R(D\text{ obs})$, applications of both these rules must be maximized. Applications of Rule 2 are maximized when the set of runs in which p_1 and p_2 are observed false completely overlap. As shown in fig. 3.3, D is observed false in $\mathbf{min}(R(\overline{p_1}), R(\overline{p_2}))$ runs. Rule 1 is applied when either p_1 or p_2 is observed true. Applications of Rule 1 are maximized when the set of runs in which p_1 and p_2 are observed true are non-overlapping. If this is possible, D is observed true in $R(p_1) + R(p_2)$ runs. In total, D is observed in $R(p_1) + R(p_2) + \mathbf{min}(R(\overline{p_1}), R(\overline{p_2}))$ runs. This is shown in Case 1 of fig. 3.3. However, there has to be an

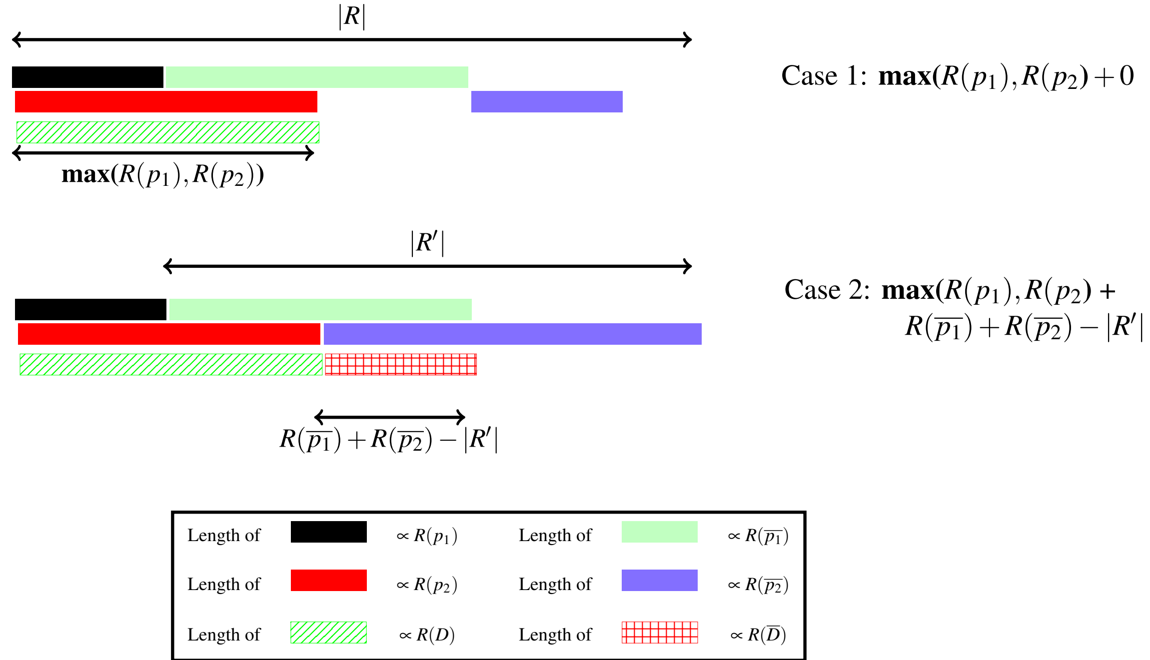


Figure 3.4: Illustration for $\downarrow R(D \text{ obs})$

overlap between true observations of p_1 and p_2 if this sum exceeds $|R|$, the total number of runs. This is shown in Case 2 of fig. 3.3. In this case, D will be observed in all of the runs. Thus,

$$\uparrow R(D \text{ obs}) = \mathbf{min(}|R|, R(p_1) + R(p_2) + \mathbf{min(}R(\overline{p_1}), R(\overline{p_2})\mathbf{))}$$

Finally, to minimize $R(D \text{ obs})$, applications of the two rules mentioned above are minimized. Per Rule 1, the true observations of p_1 and p_2 must completely overlap to minimize the runs in which D is observed true. As shown in the illustration in fig. 3.4, D is observed true in $\mathbf{max(}R(p_1), R(p_2)\mathbf{)}$ runs. This decides that p_1 and p_2 are both true in $\mathbf{min(}R(p_1), R(p_2)\mathbf{)}$ runs. This leaves $|R'| = |R| - \mathbf{min(}R(p_1), R(p_2)\mathbf{)}$ runs to pick, without overlap, the runs in which both p_1 and p_2 are simultaneously observed false. If this is possible, applications of Rule 2 are minimized.

1. Suppose, the sum $R(\overline{p_1}) + R(\overline{p_2})$ is less than $|R'|$. As shown in Case 1 of fig. 3.4, the false observations of p_1 and p_2 can be made non-overlapping. Consequently, there are no runs in which the disjunction is observed false.

Table 3.2: Bounds required in equation (3.4) for a conjunction

Quantity	Bounds for $C = p_1 \wedge p_2$
$\uparrow F(C)$	$\mathbf{min}(F(p_1), F(p_2))$
$\downarrow S(C)$	$\mathbf{max}(0, S(p_1) + S(p_2) - \lvert S \rvert)$
$\uparrow S(C \text{ obs})$	$\mathbf{min}(\lvert S \rvert, S(\overline{p_1}) + S(\overline{p_2}) + \mathbf{min}(S(p_1), S(p_2)))$
$\downarrow F(C \text{ obs})$	$\mathbf{max}(F(\overline{p_1}), F(\overline{p_2})) + \mathbf{max}(0, F(p_1) + F(p_2) - \lvert F' \rvert)$ where $\lvert F' \rvert = \lvert F \rvert - \mathbf{min}(F(\overline{p_1}), F(\overline{p_2}))$

Table 3.3: Bounds required in equation (3.4) for a disjunction

Quantity	Bounds for $D = p_1 \vee p_2$
$\uparrow F(D)$	$\mathbf{min}(\lvert F \rvert, F(p_1) + F(p_2))$
$\downarrow S(D)$	$\mathbf{max}(S(p_1), S(p_2))$
$\uparrow S(D \text{ obs})$	$\mathbf{min}(\lvert S \rvert, S(p_1) + S(p_2) + \mathbf{min}(S(\overline{p_1}), S(\overline{p_2})))$
$\downarrow F(D \text{ obs})$	$\mathbf{max}(F(p_1), F(p_2)) + \mathbf{max}(0, F(\overline{p_1}) + F(\overline{p_2}) - \lvert F' \rvert)$ where $\lvert F' \rvert = \lvert F \rvert - \mathbf{min}(F(p_1), F(p_2))$

2. If $R(\overline{p_1}) + R(\overline{p_2})$ exceeds $|R'|$, there will be a minimum overlap of $R(\overline{p_1}) + R(\overline{p_2}) - |R'|$ between the runs in which p_1 and p_2 are observed false. As shown in Case 2 of fig. 3.4, D will be observed false in those runs.

Thus,

$$\downarrow R(D \text{ obs}) = \mathbf{max}(R(p_1), R(p_2)) + \mathbf{max}(0, R(\overline{p_1}) + R(\overline{p_2}) - |R'|)$$

For simplicity and generality, the preceding discussion derives the bounds for a general set of runs R. Table 3.2 lists the specific bounds required in equation (3.4) for a conjunction $C = p_1 \wedge p_2$ by substituting the set of successful runs S or the set of failed runs F in the place of R. Table 3.3 lists the specific bounds for a disjunction $D = p_1 \vee p_2$.

As a concrete example, consider $C = p_1 \wedge p_2$. Assume that there are 1,000 successful runs, 1,000 failed runs, and that p_1 and p_2 are observed in all runs. Furthermore, assume $F(p_1) = 500$, $F(p_2) = 1000$, $S(p_1) = 250$ and $S(p_2) = 500$. Substituting $R(\overline{p}) = 1000 - R(p)$ and computing the bounds listed in table 3.2 we get $\uparrow F(C) = 500, \downarrow S(C) = 0, \downarrow F(C \text{ obs}) = 1000$ and $\uparrow S(C \text{ obs}) = 1000$. Therefore the upper bound of *Increase*(C) is $\frac{500}{500} - \frac{1000}{2000} = 0.5$ and the upper bound of *Sensitivity*(C) is $\frac{\log 1000}{\log 1000} = 1.0$.

Thus, the upper bound for $\mathit{Importance}(C)$ is

$$\frac{2}{\frac{1}{0.5}+\frac{1}{1}} = 0.666\ldots.$$

If the compound predicate C is analyzed during the iterative elimination algorithm (see section 2.3.2) and we already know a predicate C' with *Importance* score more than 0.666, we can prune C before computing its exact score. The estimate of $\mathit{Importance}(C)$ computed above is an upper-bound on the actual score of C, making it mathematically impossible for it to beat C' even if fully evaluated.

All upper- and lower- bound estimates are conservative. Pruning compound predicates using the above calculations and the appropriate threshold value reduces the computational intensity of the analysis without affecting the results. The effectiveness of pruning in practice is examined in section 3.4.4.

3.2 Usability Metric

In our experiments, we often observe hundreds of compound predicates with similar or even identical high scores. The redundancy elimination algorithm will select the top predicate randomly from all those tied for the top score; a human programmer finding a predicate to use in debugging is likely to make a similar choice. *Importance* measures predictive power, so all high-scoring predicates should be good bug predictors. However, even a perfect predictor may be difficult for a programmer to use when finding and fixing a bug.

Debugging using a simple predicate requires the identification and understanding of the connection between the predicate and the bug it predicts. For a compound predicate, the programmer must also understand the connection between its components. Given a set of compound predicates with similar high scores, those that can be easily understood by a human are preferable. However, the notion of usability is hard to quantify or measure. In this section, we propose the *effort* metric for selecting understandable predicates from a large set of high-scoring predictors. Only predicates selected by this metric are presented to the user. If the data is analyzed by an automated tool it may not be advantageous to employ this metric. The definition of effort is unrelated to a predicate's *Importance* score, making it orthogonal to the pruning

discussed in section 3.1.2. As mentioned earlier, the objective function approximated by this metric is hard to measure experimentally. However, we find that it works well in practice.

The goal of the metric is to capture the debugging effort required from the programmer to find a direct connection between component predicates. We adapt the distance metric of Cleve and Zeller [CZ05] for this purpose. They model the programmer as performing a breadth-first search in the program dependence graph while exploring the program. By this metric, the score of a predicate is the fraction of code that can be ignored while searching for the bug. We use a similar metric called *effort* for a compound predicate.

Definition 5. *The "effort" required by a programmer while using a compound predicate $\phi(p_1, p_2)$ is the smaller fraction of the code explored in a breadth-first bidirectional search for p_2 from p_1 and vice-versa.*

The idea behind this metric is that the larger the distance between the two predicates, the greater the effort required to understand their relationship. Also, if a large number of other branches are seen during the search, the programmer should keep track of these dependencies too. Per Cleve and Zeller [CZ05], we use the program dependence graph (PDG) to model the program rather than the source code. We perform a breadth-first search starting from p_1 until p_2 is reached and count the total number of vertices visited during the search. The fraction of code covered is the ratio of the number of visited vertices to the total number of PDG vertices.

The *effort* metric can be applied both *proactively* or *reactively*. Proactive use removes compound predicates whose *effort* values fall above a certain threshold of usefulness. This avoids computing their scores and hence improves performance. Reactive use of the metric breaks ties among predicates with the same *Importance* score by giving higher ranks to those with better *effort* value.

Our definition of *effort* is meant to be exhaustive: if there is any possibility of two predicates being related using less than the chosen percentage of nodes, conjunctions and disjunctions of them needs to be considered. While pruning compound predicates, an exhaustive metric is preferable. However, it is not quite clear if this metric can be meaningfully incorporated into statistical analysis. We can use it to break ties in the *Importance* scores. As mentioned in section 2.4, it is hard to qualitatively evaluate the relevance of a single ranked list of predicates, let alone the marginal benefits of ranked lists of predicates from different techniques.

The *effort* metric is just one of many reasonable candidates. The following are some other options.

- A *selective* metric can eschew bi-directional search of the PDG in favor of a directed forward-only or backward-only search.
- When the failing locations are available, the programmer might evaluate the influence of a predictor on such locations. A *chop* [JR94; RR95] between the predictor and the failure location can characterize this activity.

Manual evaluation of the ranked list of predicates is needed to understand the relative merits of each candidate metric. We manually evaluate the usefulness of compound predicates in section 3.4.2. In addition to being subjective, manual evaluation is time consuming, and needed 10 hours for the 130 Siemens programs. To avoid the subjectivity of evaluation by a single person, Parnin and Orso [PO11] conduct a study on how 34 participants (who were graduate students) used the output of the Tarantula [JH05] debugging tool. While they find some interesting patterns on how the tool was actually used, further studies involving practicing programmers are needed to reliably characterize the usefulness of fault-localization results.

3.3 Case Studies

This section presents two case-studies where compound predicates prove to be useful. The first study concerns a memory access bug in `exif` 0.6.9, an open source image manipulation program. A compound predicate is useful in increasing the score of an extremely useful bug predictor. In the second case study, a compound predicate is useful in isolating a bug in the `print_tokens` program in the Siemens bug benchmark [Hut+94].

3.3.1 `exif`

`exif` 0.6.9 crashes while manipulating thumbnails in images taken using Canon cameras. The module handling Canon images has a bug in the function `exif_mnote_data_canon_load`. A snippet from this function is shown in fig. 3.5.

```
for (i = 0; i < c; i++) {
    ...
    n->count = i + 1;
    ...
    if (o + s > buf_size) return; // (a)
    ...
    n->entries[i].data = malloc(s); // (b)
    ...
}
```

Figure 3.5: Code snippet from function `exif_mnote_data_canon_load` in `exif`

Table 3.4: Results for `exif` with only simple predicates

Score	Predicate	Function	File:Line
0.704	new value of `len` == old value of `len`	`jpeg_data_load_data`	jpeg-data.c:224
0.395	i == s	`exif_mnote_data_canon_save`	exif-mnote-data-canon.c:176

If the condition `o + s > buf_size` is true on line (a), then the allocation of memory to the pointer `n->entries[i].data` on line (b) is skipped. The program crashes when another piece of code reads from `n->entries[i].data` without checking if the pointer is valid. This is an example of a non-deterministic bug as the program succeeds as long as the uninitialized pointer is not accessed somewhere else.

We generated 1,000 runs of the program using input images randomly selected from a set of Canon and non-Canon images. As the bug being studied rarely manifests, this set of images was designed to trigger sufficient failed executions. Each run was executed with randomly-generated command line arguments, omitting arguments that would have triggered two unrelated bugs. There are 934 successful executions and 66 crashes. Applying the iterative bug isolation algorithm with only simple predicates produces two predicates that account for all failed runs as shown in table 3.4. Studying the source code of the program does not show any obvious relation between the two predictors and the cause of failure. Although the second predictor is present in the crashing function, it is a comparison between two unrelated variables: the loop iterator `i` and the size of the data stored in the traversed array `s`. Also it is true in only 31 of the 66 failures.

The analysis assigns a very low score of 0.019 to the predicate p_1: `o + s > buf_size` despite the

Table 3.5: Results for `exif` with compound predicates

Score	Predicate	Function	File:Line
0.941	`o + s > buf_size`	`exif_mnote_data_canon_load`	exif-mnote-data-canon.c:237
	and `offset < len`	`exif_data_load_data`	exif-data.c:644

fact that it captures the exact source of the uninitialized pointer. Because the bug is non-deterministic, p_1 is also true in 335 runs that succeeded, making p_1 a partial predictor. In other words, p_1 is only a prerequisite of failure: it is observed true in all failed runs, but it is also observed true in some or all successful runs. Including compound predicates in the analysis produces one compound predicate shown in table 3.5. The second row is the second component of a compound predicate, which is a conjunction as indicated by the keyword *and* at the start. Conjunction of p_1 with the second predicate p_2: `offset < len` eliminates all false positives and thereby earns a very high score. This is an example of how a conjunction can improve the score of a partial predictor. p_2 is in function `exif_data_load_data`, which calls `exif_mnote_data_canon_load` indirectly. It is possible that p_2 is another partial predictor, capturing another condition that drives the bug to cause a crash. If it does, it has to be a deep relationship as we could not find such a relation even after spending a couple of hours trying to understand the source code. However this does not reduce the importance of this result as the conjunction has a very high score compared to p_1 and p_2 individually.

A hypothetical predicate p_3: `n->entries[i].data == 0` at the point where the uninitialized pointer is actually used ought to be a perfect bug predictor. However, the CBI instrumenting compiler does not actually instrument this condition or any direct equivalent. Furthermore, this assumes that `n->entries[i].data` is zero-initialized even when `exif_mnote_data_canon_load` returns early without filling in this field. Predicate p_1 provides critical additional information, as it identifies the initial trigger (skipping the `malloc`) that sets the stage for eventual failure (use of an uninitialized pointer). Thus one role for compound predicates is to capture those program behaviors, like p_1, that are necessary but not sufficient preconditions for failure.

It should be noted that the compound predicate shown in table 3.4 is one of several top-ranked predicates. The iterative bug isolation algorithm used by CBI picks a random top-ranked predicate when

```
switch(next_st)
 {
     ...
     case 30 : /* This is COMMENT case */
               skip(tstream_ptr->ch_stream);
               /* missing code token_ind= */ next_st=0; // (a)
               break;
 }
```

Figure 3.6: Code snippet from function `get_token` in `print_tokens`

Table 3.6: Results for `print_tokens` with simple predicates

Score	Predicate	Function	File:Line
0.609	`token_ind >= 80`	`get_token`	print_tokens.c:195
0.091	`token_ptr->token_id == 27`	`print_token`	print_tokens.c:508

Table 3.7: Results for `print_tokens` with compound predicates

Score	Predicate	Function	File:Line
0.733	`next_st == 30` *and* new value of `token_ind` < old value of `token_ind`	`get_token` `get_actual_token`	print_tokens.c:219 print_tokens.c:557
0.691	`next_st == 30` *and* `token_ind > cu_state`	`get_token` `get_token`	print_tokens.c:219 print_tokens.c:213
0.592	`next_st == 30` *and* `token_ind == cu_state`	`get_token` `get_token`	print_tokens.c:219 print_tokens.c:197
0.474	`next_st == 30` *and* `start < ind`	`get_token` `get_actual_token`	print_tokens.c:219 print_tokens.c:561
0.474	`next_st == 30`	`get_token`	print_tokens.c:219

there are several predicates with the same top score. This demonstrates the need for further studies to quantify the usability of compound predicates in addition to the metric in section 3.2.

3.3.2 `print_tokens`

The second case study is about the bug in the function `get_token` in `print_tokens v6`. Figure 3.6 shows a snippet from this function. The bug is the omitted assignment to variable `token_ind` in line (a).

Table 3.8: Properties of applications in the Siemens test suite

Application	Variants	LOC	Sites	Test Cases
`print_tokens`	7	727	268	4,130
`print_tokens2`	10	569	250	4,115
`replace`	31	563	567	5,542
`schedule`	9	413	229	2,650
`schedule2`	9	373	265	2,710
`tcas`	41	173	496	1,608
`tot_info`	23	564	296	1,052

The output of the iterative bug isolation algorithm with simple predicates is shown in table 3.6. The top predictor is in the same function as the bug, and tests the value of the variable `token_ind` whose value is affected by the bug. But, it is hard to gleam any information beyond that. It is also the case that the predicate *p*: `next_st == 30` at the **`switch`** statement is an insufficient case of failure. In runs where the value of `token_ind` reaching line (a) was already `0`, there is no failure. Among 3,868 successful and 150 failed runs overall, *p* was true in 1,250 successful and all 150 failed runs. Hence, *p* had a score of 0.095 and was not presented to the user.

Table 3.7 shows the output of bug isolation with compound predicates. There are four compound predicates and one simple predicate in the result. The common thread among all these predicates is the predicate *p*. This hints to the programmer to scrutinize the statements in the **`case`** `30` block of the **`switch`** statement. The second component in the top three compound predicates checks the values of the `token_ind` variable. This further hints to the programmer that the variable `token_ind` is wrongly assigned. Thus, compound predicates can, potentially, help a programmer experienced with the code base to quickly identify the bug and develop a fix.

3.4 Experiments

This section presents a quantitative evaluation of the ideas presented in previous sections. The experiments are performed using the Siemens test suite [Hut+94] as maintained by the Galileo Software-artifact Infrastructure Repository (SIR) [Rot+06]. Each application in the Siemens test suite has multiple variants,

Table 3.9: Kind of the top predicate during complete data collection

		Type of Top Predictor		
Application	Variants	Simple	Conjunction	Disjunction
`print_tokens`	7	0	7	0
`print_tokens2`	10	0	10	0
`replace`	31	3	28	0
`schedule`	9	0	0	9
`schedule2`	9	1	8	0
`tcas`	41	1	40	0
`tot_info`	23	2	20	1
Overall	130	5%	86%	7%

each with a seeded bug, and an associated suite of test inputs. The seeded bugs cause the programs to print a wrong output, instead of making them crash. Each test case is labeled a success or a failure by comparing the output of the buggy program to that of a bug-free reference version. Table 3.8 lists the number of variants, the size in LOC, the number of instrumentation sites, and the size of their test suite. We report aggregate results by averaging the relevant measures across all variants of each application. We use CodeSurfer [Gra06] to build program dependence graphs and to compute the *effort* metric. There are two configurable parameters for the experiments: the sampling rate and the *effort* cutoff. Unless specified, the default sampling rate is 1 (i.e., complete data collection) and the default *effort* is 5% (i.e., only predicates that are reachable from each other by exploring less than 5% of the program are considered).

3.4.1 Top-scoring Predicates

For the 130 buggy programs in the Siemens suite, we perform a statistical debugging analysis using the iterative bug isolation algorithm discussed in section 2.3, at a sampling rate of 1. We then determine whether the top-scoring bug predictor is a simple predicate, a conjunction, or a disjunction. Table 3.9 reports how often each of these three kinds of predictors appears with the highest score.

As can be seen, conjunctions dominate, with 86% of programs tested selecting a conjunction as the top bug predictor. This confirms that compound predicates can diagnose failures more accurately than simple predicates alone. If there are multiple bugs in the program, disjunctions of high-scoring predicates for two

Table 3.10: Kind of the top predicate during 1⁄100 sampling

Application	Variants	Type of Top Predictor		
		Simple	Conjunction	Disjunction
print_tokens	7	0	7	0
print_tokens2	10	0	10	0
replace	31	7	24	0
schedule	9	2	7	0
schedule2	9	5	4	0
tcas	40	31	9	0
tot_info	21	0	21	0
Overall	127	35%	64%	0%

bugs can be expected to have high scores. Because each Siemens program variant has only a single bug, it is to be expected that disjunctions are not needed as frequently. Thus, disjunctions play a smaller but significant role, especially in the case of schedule. Even in single-bug programs, disjunctions can be helpful if no one simple predicate perfectly aligns with the condition that causes failure.

Section 3.4.5 explains that the chance of observing a compound predicate shrinks quadratically with the sampling rate. However, conjunctions remain important even with sparse sampling. Table 3.10 reports how often each kind of predicate appears with the highest score while sampling at a rate of 1⁄100. Only 64% of the applications have a conjunction as the top-scoring predictor. When tcas is excluded, 84% of the Siemens applications show a conjunction as the top-scoring predictor. The tcas program has no loops or recursion and every function is called at most twice. Thus, no simple predicate can be observed more than twice in a single tcas run, and conjunctions are infrequently observed when sampling is sparse. Only 25% of the tcas experiments have a compound predicate as the top-ranked predictor.

3.4.2 Bug-relevance of Compound Predicates

In the previous subsection, we evaluated the effectiveness of compound predicates based on their failure-predictive ability. In this section, we perform a qualitative evaluation of their utility in bug isolation. The objective is to provide a qualitative substantiation of the results in the previous subsection: does the higher failure-predictivity of compound predicates, as captured by the *Importance* metric, actually matter?

Table 3.11: Bug-relevance of simple and compound predicates

	Compound Predicates		
Simple Predicates	Irrelevant	Not-obvious	Relevant
Irrelevant	34	12	21
Not-obvious	0	12	15
Relevant	10	6	20

For this evaluation, we consider each program in the Siemens suite, and compare the output of analyses performed with and without compound predicates. We subjectively, with prior knowledge of the location of the bug, classify the output into three classes:

- *Irrelevant*: no direct connection between the predicate and the bug is evident.
- *Not-obvious*: no direct connection between the predicate and the bug is evident, but the predicate is very close to the location of the bug. A programmer experienced with the code base may be able to find such predicates useful.
- *Relevant*: the predicate is relevant to the bug, and captures a very obvious pre-condition or post-condition.

For variants where simple predicates are relevant, compound predicates are considered relevant only when they are strictly more useful than simple predicates.

This is one, albeit subjective, way of evaluating the usefulness of compound predicates. The manual evaluations in this subsection required about 10 hours. Table 3.11 shows the result of this classification. The rows capture the classification of analyses performed with only simple predicates, and the columns capture the classification of analyses performed with compound predicates. The cells count the number of variants that fall in each category. The right-most column (including the bottom-right cell, because of our stricter evaluation when simple predicates are labeled relevant) represents cases where compound predicates are strictly more useful than simple predicates. We also consider cases where simple predicates are labeled irrelevant and compound predicates are labeled not-obvious as cases where compound predicates are strictly more useful. Among the 130 variants, compound predicates are strictly more useful for 68 variants. Compound predicates received the same categorization as simple predicates for 46 of the remaining 62

Table 3.12: Bug-relevance of top-ranked compound predicates found with and without *effort* metric

	Without *effort*		
With *effort*	Irrelevant	Not-obvious	Relevant
Irrelevant	40	0	4
Not-obvious	2	25	3
Relevant	8	1	47

variants. Compound predicates received a worse categorization for the remaining 16 variants. This includes the 7 variants in table 3.9 where the top-ranked simple predicate had a higher score than the top-ranked compound predicate. In these seven cases, incorporating compound predicates does not negatively impact bug isolation, since the top-ranked compound predicate is not presented to the user. In summary, based on both quantitative and qualitative evaluation, incorporating compound predicates is more useful than just simple predicates.

Relevance of Usability Metric

We perform a similar evaluation to test the impact of the *effort* heuristic in the bug-relevance of compound predicates. We classify the top-ranked compound predicates found when the *effort* metric is used to prune predicates, and when all the compound predicates are considered. The latter case captures the penalty of using *effort* to prune predicates. This classification was done blindly: when classifying a compound predicate, whether this predicate was found while *effort* is used is deliberately omitted.

Table 3.12 shows the result of this classification. In 77 of the 130 variants, the top-ranked compound predicate was the same irrespective of the usage of *effort*. Even when the top-ranked compound predicates differed, they had similar categorization for most of the variants. The classification differed in only 28 of the 130 variants. Using *effort* discarded bug-relevant predicates in 7 variants. However, *effort* was able to prune irrelevant, but high-scoring predicates, and find relevant predicates in 11 variants. Our motivation in the design of *effort* is to design an exhaustive metric: if there is any way two predicates could be related using less than the chosen percentage of nodes, a compound predicate of them needs to be considered. A selective metric that eschews bi-directional search of the PDG in favor of a directed search may perform better in selecting the most bug-relevant compound predicates.

Table 3.13: Properties of larger bug benchmarks

Application	Variants	LOC	Sites	Test Cases
`flex`	18	14,725 to 16,967	1,491 to 11,704	567
`grep`	17	14,775 to 15,721	25,701 to 40,167	809
`gzip`	9	7,348 to 8,970	27,822 to 35,915	217
`sed`	15	8,704 to 19,774	16,090 to 19,030	365

Table 3.14: Kind of the top predicate for large bug benchmarks

		Type of Top Predictor		
Application	Variants	Simple	Conjunction	Disjunction
`flex`	18	3	15	0
`grep`	17	1	16	0
`gzip`	9	2	7	0
`sed`	15	1	14	0
Overall	59	12%	88%	0%

3.4.3 Experiments on Larger Benchmarks

In this section, we briefly evaluate compound predicates on a set of larger benchmarks. We use the following bug benchmarks obtained from the Software-artifact Infrastructure Repository [Rot+06], all written in C: `flex`, `grep`, `gzip`, and `sed`. Each program has multiple versions and each version has multiple faults. Some of the faults do not cause significantly many test cases to fail. Since a sizable sample of failed runs is needed for statistical debugging to be effective, we retain only those faults that cause more than five test cases to fail. Table 3.13 shows the the total number of faulty variants across all versions of each program that cause more than five test cases to fail. The table also shows the size of the programs in terms of LOC, the number of intrumentation sites, and the size of the test suite.

For the 59 buggy programs in this study, we perform statistical analysis on data collected at a sampling rate of 1. The *effort* parameter is set to 5%. We determine whether the top-scoring bug predictor is a simple predicate, a conjunction, or a disjunction. Table 3.14 reports how often each of these three kinds of predictors appears with the highest score. A conjunction was the top-scoring predictor in 88% of the programs tested. A simple predicate was the top-scoring predictor for the remaining 12% of the programs.

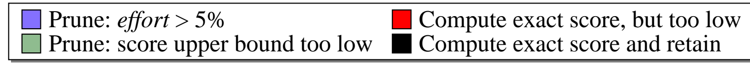
Table 3.15: Time (in minutes) taken to compute scores for compound predicates

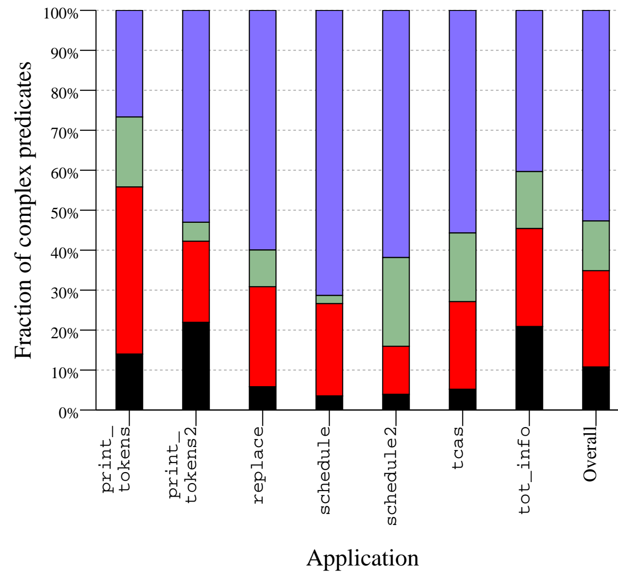
Application	Naïve	*effort*	*effort* + *Importance*	*effort* + Only-Best
`print_tokens`	139.95	64.17	39.60	6.53
`print_tokens2`	19.16	7.16	2.88	0.69
`replace`	37.04	14.73	10.33	2.35
`schedule`	3.01	1.10	0.67	0.06
`schedule2`	4.38	1.32	0.50	0.13
`tcas`	6.24	3.15	1.41	0.33
`tot_info`	3.59	1.69	1.40	0.16
Overall	20.95	8.98	5.59	1.11

A disjunction was never a top bug predictor. This result confirms the conclusion from experiments on the Simens programs that compound predicates can diagnose failures more accurately than simple predicates alone.

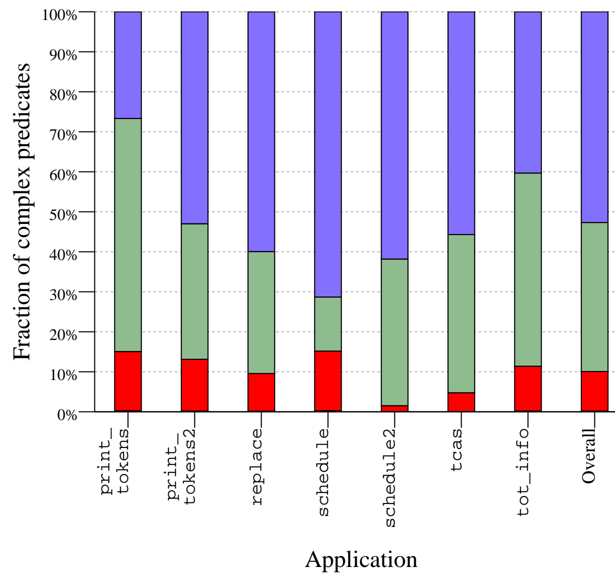
3.4.4 Effectiveness of Pruning

Even when restricted to binary conjunctions and disjunctions, compound predicates could substantially increase the analysis workload if naïvely implemented. Techniques to reduce the computation time were proposed in earlier sections. Table 3.15 shows their benefits. For each application, the first column shows the time to naïvely compute the scores of compound predicates. Section 3.2 suggests a heuristic for pruning compound predicates that are unlikely to be useful or understandable to a programmer. The "*effort*" column shows the analysis time with this metric enabled. Section 3.1.2 describes how to compute an upper bound on a predicate's score. Two thresholds can be used while predicates are pruned based on their upper bound. First is the *Importance* score of the component simple predicates. The column titled "*effort* + *Importance*" shows the analysis time with this threshold along with the *effort* metric. When interested in only the score of the top compound predicate (such as section 2.3.2), the threshold of the highest *Importance* score found so far can be used as the threshold. The "*effort* + Only-Best" column shows the analysis time for this aggressive threshold. Each optimization progressively reduces the running time, realizing an improvement from 21 minutes to just a minute. With the naïve setting, the `print_tokens` application takes more than two hours to complete. Even with aggressive pruning, the analysis takes





(a) Conjunctions



(b) Disjunctions

Figure 3.7: Percentage of predicates pruned using *effort* and upper-bound in *Importance*

almost 7 minutes. This is due to the relatively low effect of the *effort* and pruning optimizations on the `print_tokens` program, as seen in Figure 3.7.

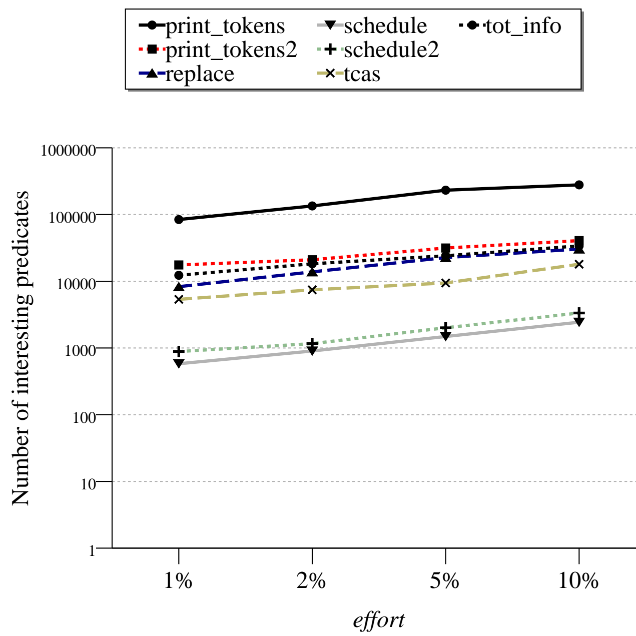
Figure 3.7 shows the relative efficiency of the *effort* metric and pruning in discarding predicates. On average, 53% of candidate conjunctions and disjunctions are discarded because, per the *effort* metric, they would require traversing more than 5% of the application code. Pruning conjunctions whose upper bound of the *Importance* scores are lower than the scores of their constituent simple predicates (the $\mathit{effort} + \mathit{Importance}$ setting in table 3.15) eliminates a further 15% of the conjunctions. This step is more useful for disjunctions: 37% of the disjunctions were pruned away. Only 22% of conjunctions and 10% of disjunctions remain to have their exact scores computed. Of this, roughly a fourth (6% of the initial pool) of the conjunctions have scores greater than their component predicates. Less than 1% of the disjunctions had scores greater than their component predicates. The main take-away from this section of experiments is that pruning and the *effort* heuristic reduce the computation time from 21 minutes on average to just over a minute.

3.4.5 Effect of Effort and Sampling

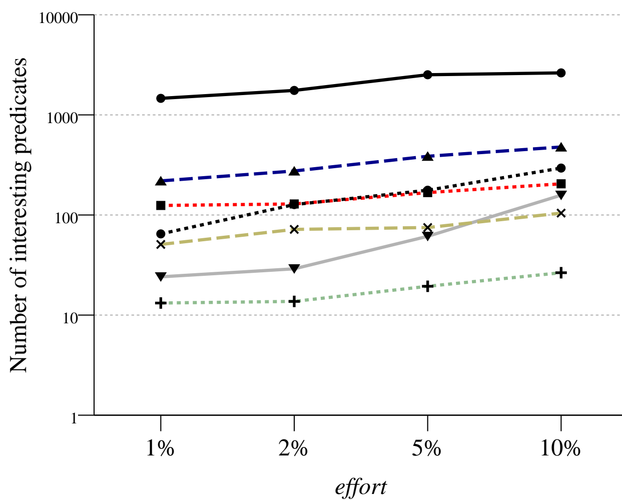
Figure 3.8 shows how the number of interesting conjunction and disjunction predicates, as defined in Definition 4, varies at four different *effort* levels. As expected, more predicates are evaluated as *effort* increases, and so more interesting predicates are found.

Real deployments of CBI use sparse random sampling of simple predicates to reduce performance overhead and protect user privacy. Prior work [Lib+03] has recommended sampling rates of $1/100$ to $1/1{,}000$ to balance data quality against performance and privacy concerns. However, a pair of independent features each observed in $1/100$ runs have only a $1/10{,}000$ chance of being observed together, raising doubts whether interesting compound predicates will be found in sparsely sampled data. Note, however, that certain compound predicate values can be “observed” even if one simple component is not, per the rules in table 3.1.

Figure 3.9 shows the effect of sampling on the number of interesting predicates of each kind. We term a simple predicate as interesting if its *Importance* score is strictly greater than 0. A compound predicate is



(a) Conjunctions



(b) Disjunctions

Figure 3.8: Variation in the number of interesting predicates with *effort*

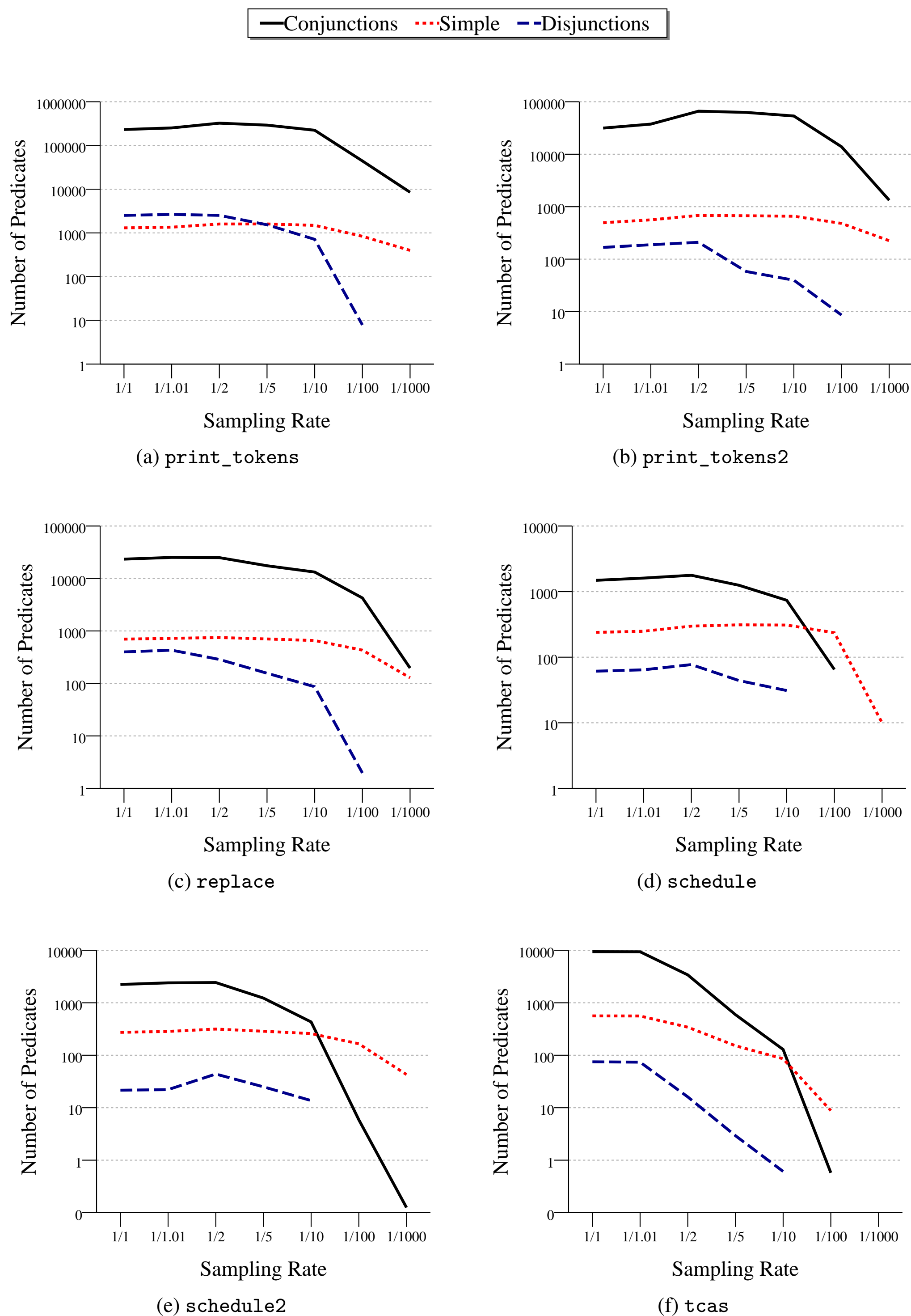
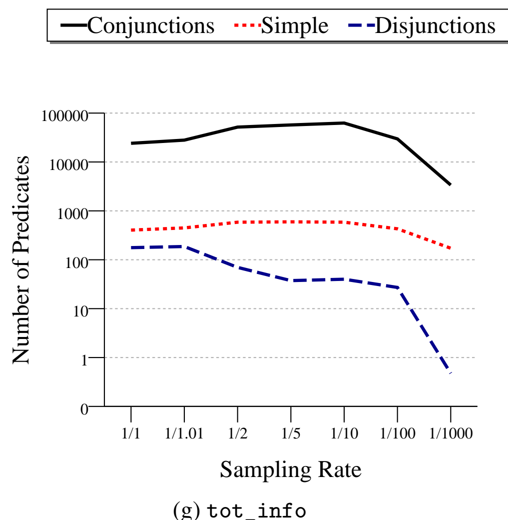


Figure 3.9: Sampling rate vs. number of interesting predicates



(g) `tot_info`

Figure 3.9: Sampling rate vs. number of interesting predicates (cont.)

considered interesting based on Definition 4. The sampling rates are reduced exponentially from 1⁄1 (i.e. no sampling) to 1⁄1,000. Figure 3.9 has one plot per Siemens application showing the variation in the number of interesting predicates as the sampling gets sparser. Note that the *y* axes have a logarithmic scale, and the scale is not the same across applications.

The number of interesting disjunctions is always very low (order of tens) compared to interesting conjunctions. However, disjunctions should not be omitted altogether. They are useful in some programs, such as `schedule` (table 3.9). Pruning is very effective for disjunctions, and hence they do not impose as high an overhead as conjunctions. At sampling rates lower than 1⁄10, there is a sharp drop in the number of interesting conjunctions. This is likely due to the shrinking odds of observing both components of a conjunction within a single run. Despite the sharp drop, the number of interesting conjunctions is still comparable to the number of interesting simple predicates even at 1⁄1,000 sampling. This shows that interesting compound predicates can still be found at sparse but realistic sampling rates.

A puzzling trend in fig. 3.9 is that all three curves rise for a brief interval before dropping off. Most Siemens applications exhibit this bump. The bumps in the conjunction and disjunction curves could be attributed to the bump in the simple predicates curve. Any increase in the number of interesting

simple predicates is likely to produce a greater increase in the number of interesting compound predicates, especially because the additional simple predicates are likely to be redundant (as explained later).

The transient increase in the number simple predicates at moderate sampling rates is unexpected and initially seems counterintuitive. Closer inspection of experimental results reveals two scenarios where this can happen.

The first scenario happens due to an ad hoc but perfectly reasonable elimination of seemingly identical predicates. As an example, consider the predicates p: `a == b` and q: `a >= b` defined at the same site. Note that q is a derived predicate (see section 2.1), and is seen in runs in which either the predicate $a == b$ is true or $a > b$ is true. If both p and q have the same score, it is useful to just retain p as it is a more stringent condition than q. However, this does not mean that `a > b` was never true. It may be the case that `a > b` was observed during some runs but does not affect the outcome of `a >= b` if `a == b` also happens to be true in those runs. However, at sampling rates lower than 1, only `a > b` may be observed in some runs and hence the number of runs in which p and q are observed true, and consequently their scores, may be different. Thus, the ad hoc elimination heuristic performs less effectively at lower sampling rates, leading to an increase in the number of simple predicates retained for further analyses.

To understand the second scenario, consider a predicate p for which $S(p \text{ obs}) = S(p)$ and $F(p \text{ obs}) = F(p)$. In other words, p is true at least once in every run in which it is observed. From equation (3.2), $\mathit{Increase}(p) = 0$. In a run in which p was observed, it may also be false at least once. As we reduce the sampling rates, only the false occurrences may be recorded in some runs and hence the two equalities may no longer hold. As a result $\mathit{Increase}(p)$ may be nonzero. If it becomes positive, then a predicate that was not interesting at higher sampling rates becomes interesting at lower sampling rates.

3.5 Summary

We have demonstrated that compound Boolean predicates are useful predictors of bugs. Our experiments show qualitative and quantitative evidence that statistical debugging techniques can be effectively applied to compound predicates, and that the resulting analysis provides improved results. We describe two methods of eliminating predicate combinations from consideration, making the task of computing compound

predicates more feasible. The first employs three-valued logic to estimate set sizes and thereby estimate the upper bound of the score of a compound predicate. The second uses distances in program dependence graphs to quantify the programmer effort involved in understanding compound predicates.

These techniques help the statistical debugging analysis scale up to handle the large number of candidate predicates we consider. However, using the analysis results in debugging can require sifting through a large number of less useful predicates that also pass automated inspection. Further shrinking this list while retaining useful predictors remains an important open problem. Most identified bug predictors redundantly describe the same small set of program failures. Thus the bi-clustering algorithm of Zheng et al. [Zhe+06] may be promising as it was designed to handle multiple predictors for the same bug. Automated analyses which further process predictor lists, such as BTrace [Lal+06], may also benefit from the richer diagnostic language offered by the work presented here.

Chapter 4

Adaptive Bug Isolation

As mentioned in chapter 3, CBI gathers feedback reports by using valuable CPU cycles at end user machines. It is essential to maximize the utility of these CPU cycles. Statistical debugging, however, has to cast a wide net of instrumentation to find useful failure predictors. This chapter presents a feedback-driven technique that prioritizes data collection towards those predicates that are most likely to be useful in fault localization. The advantages of this technique are three-fold. First, instrumentation of predicates that are not failure-predictive can be avoided, or removed after their predictivity is ascertained. Second, this technique has a configurable parameter that controls the amount of instrumentation performed. At the most aggressive setting of this parameter, mean overheads of $1.03\times$ are achievable, while still producing useful results for fault localization. Third, sampling is no longer required to guarantee low-overheads. Using adaptive instrumentation, and complete (non-sampled) observation of instrumentation sites, we can quickly gather sufficient data where it is most needed.

CBI currently performs broad-spectrum instrumentation of numerous program behaviors. While this seems necessary to catch a wide variety of bugs, it guarantees that almost all the collected data is uninteresting. Most programs mostly work: nearly all code in any given application is not relevant for any given bug. In one study, fewer than 1 in 25,000 instrumented behaviors were reported as failure-predictive [Lib+03]. Over 99.996% of each execution profile was discarded, but only after consuming resources (CPU time, network bandwidth, storage space, etc.) that could have been better-used for other purposes.

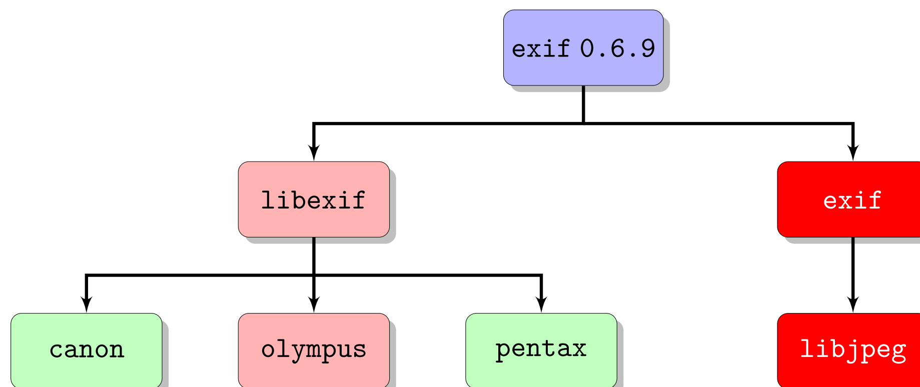


Figure 4.1: Module structure in exif 0.6.9

As a concrete example, let us return to the exif program studied in chapter 3. Figure 4.1 shows the high-level organization of exif. The core functionality to manipulate meta-data tags in images is organized in the libexif library. It has three submodules, canon, olympus, and pentax, each handling photos taken with a different brand of camera. The exif module, which is a sibling of libexif, provides a command-line interface to libexif. It has a submodule, libjpeg to manipulate images in the jpeg file format. There are three bugs in exif that are exercised by our test suite. The bug in the canon module, mentioned in chapter 3, is very rare and causes just one failure in our test suite of 10,000 runs.

The other two bugs are in the exif and libjpeg submodules. Statistical analysis finds bug predictors in the libexif, olympus, exif, and libjpeg modules. The bug in libjpeg arises when processing images with fewer than two meta data sections. The predictors for this bug are in the libjpeg, libexif and olympus modules. The single predictor in the olympus is a weak predictor. The other bug is in the printing code in exif. Good predictors for this bug are found in the exif and libexif modules. Only 0.03% of predicates in these modules are failure predictive. More importantly, none of the predicates in canon and pentax are failure predictive. This is not surprising because the code in these modules has no relevance to the known bugs. The CPU time, network bandwidth and storage space used to instrument and analyze predicates in such bug-free code are wasted. The adaptive bug isolation technique developed in this chapter aims to avoid instrumenting such bug-free code segments, but without any knowledge of the bugs. It uses feedback from deployed runs to achieve this task.

The above example highlights the problem with current monitoring systems. They begin with the

worst-case assumption that nearly anything could be a clue for a bug. Moreover, they continue monitoring events even after statistical analyses show that most are not predictive of failure. Contrast this with the focused debugging activity of an expert programmer. Using feedback from a prior execution, or even just an initial hunch, the programmer uses breakpoints and other probes near points of failure to get more feedback about program behavior. Suspect code is examined more closely, while irrelevant code is quickly identified and ignored. Each iteration enriches the programmer's understanding until the reasons for failure are revealed.

We propose to mimic and automate this process on a large scale. Instead of a single run, we can collect feedback from thousands or millions of executions of the program by its users. Adaptive bug isolation starts by monitoring a small set of program behaviors. Based on analysis of feedback obtained during this stage, other behaviors that could be causing failures are automatically chosen and monitored during the next stage. Throughout this process, statistical-analysis results are available to the programmer, who can fix failures if enough data is available or choose to wait for more data if the picture is unclear. Effectively, we replace sampled measurement of all predicates with non-sampled measurement of adaptively-selected predicates. Non-sampled instrumentation allows faster adaptation by quickly gathering sufficient data where it is most needed. Adaptive instrumentation improves upon existing approaches by prioritizing the monitoring of potentially useful behaviors over those that are less useful, thereby conserving computational resources and bandwidth for both users and developers.

The rest of the chapter is organized as follows. Section 4.1 provides a high-level overview of the changes we propose to CBI. It proposes binary instrumentation as a mechanism to efficiently change instrumentation after deployment. Section 4.2 describes our binary instrumentor and several optimizations developed to reduce the overhead of monitoring. Section 4.3 describes our Adaptive Bug Isolation technique and section 4.4 presents an experimental evaluation.

4.1 Overview of Adaptive Bug Isolation

Figure 4.2 shows a high-level overview of adaptive bug isolation. The proposed changes to the overview of CBI in fig. 1.1 are highlighted with a dark shadow. In this system, the program is instrumented to

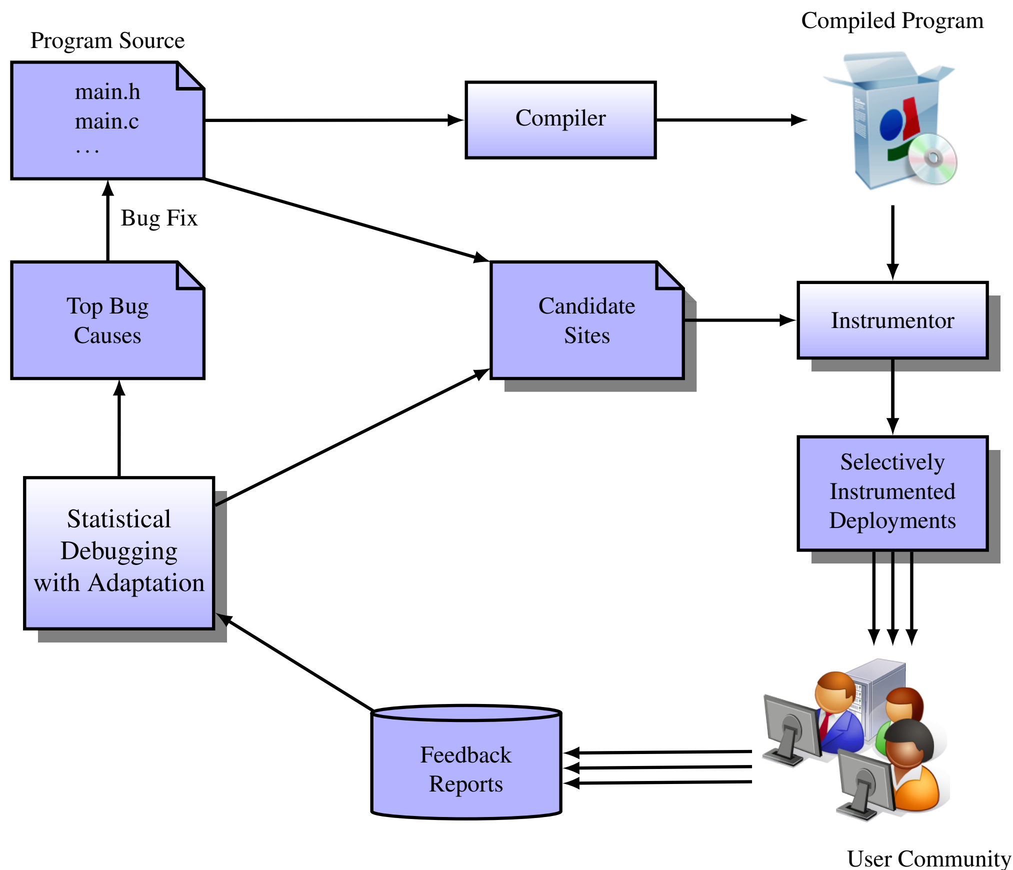


Figure 4.2: Overview of adaptive bug isolation (a dark shadow highlights differences from fig. 1.1)

selectively monitor a set of candidate instrumentation sites. The instrumentor is represented separately from the compiler to capture a design issue explored shortly. The candidate sites are initially selected statically from the program source. The number of locations instrumented is a small fraction of the whole program. Feedback reports from this selectively instrumented program are collected, as earlier. The adaptation step analyzes these feedback reports to produce a new set of candidate instrumentation sites. During any stage of this iterative process, intermediate results of fault localization are also available for the programmer's inspection. The overhead of monitoring is low, since only a small number of sites are instrumented during any iteration. The price of such low overheads is a drastic reduction in the amount of data available for statistical analysis. It is impossible to improve upon CBI's fault localization under these constraints. Our technique is, instead, geared towards retaining the fault localization ability of existing

statistical analyses, while also achieving low overheads, and avoiding instrumentation of bug-free code.

A key design issue in the design of adaptive bug isolation is the mechanism of changing instrumentation after deployment. Application sizes and bandwidth limits preclude releasing new software each time the adaptive bug-hunting system identifies new instrumentation targets. Distributing executable patches might be impractical, and resource intensive. Instead, we use binary instrumentation as the chief mechanism for adaptivity. Adaptation decisions are distributed as a list of predicates that need to be enabled or disabled. A binary instrumentor at the user's machine re-instruments the software every time a new list is distributed. Due to limitations in our binary instrumentor, it cannot support the scalar-pairs instrumentation scheme. So, all the experiments in this chapter, including non-adaptive, complete instrumentation performed using source-to-source instrumentation, are performed without the scalar-pairs instrumentation scheme.

Binary instrumentation in itself has several advantages over source-level instrumentation. We can instrument any program, not just those written in languages supported by a source-level instrumentor. We can instrument and monitor a program even when its source code is unavailable. We can also instrument system and third-party libraries used by the application. Truly fixing bugs without source code would be difficult, but remediation may still be possible once the causes of failure are identified [LS05a; LS05b; NS05; Qin+07; Rei+06; Wan+04]. Furthermore, because sampling is not used, binary instrumentation adds only a constant overhead to the size of distributed software: the size of one generic binary instrumentor, usable for all monitored software in a machine. This improves on static sampling schemes whose fast- and slow- path code variants roughly double the size of executables [Lib+03], thereby increasing costs for packaging or network distribution.

4.1.1 Practical Considerations

While the adaptive bug isolation technique we develop in this chapter is not ready to be deployed to the end users yet, we briefly consider some concerns that may arise when this system is put into practice.

Table 4.1: Taxonomy of changes, with the proposed system being the bottom right cell

	Pre-deployment Instrumentation	Post-deployment Instrumentation
Fixed Instrumentation	+ compiler optimizations – big executables	+ no source needed, ease of deployment – no compiler optimizations, high overheads
Adaptive Instrumentation	+ compiler optimizations small executables – costly patch deployment	+ easy deployment, small executables, low overheads – no compiler optimizations

Mechanism of Instrumentation

We proposes changes to two aspects of CBI: the choice of predicates, and the instrumentation mechanism. First, adaptive instrumentation changes instrumentation, even after deployment. Second, post-deployment, binary instrumentation is used in lieu of pre-deployment, source-level instrumentation. Table 4.1 shows the relative merits of each of these decisions.

1. Pre-deployment instrumentation is done at the source level, and consequently, compiler optimizations are done after the instrumentation is added. This can reduce the overhead of the instrumented code. Fixed, pre-deployment instrumentation, however, relies on sampling to reduce overheads. This results in big executables, and wasted data collection.

2. Fixed, post-deployment instrumentation increases the size of the binary after deployment, thereby reducing distribution overheads. Section 4.2 describes our exploration of this option. It evaluates several binary instrumentation toolkits. However, without adaptive instrumentation, their overheads are quite high (section 4.2.4).

3. Pre-deployment adaptive instrumentation allows small executables, and low overheads. However, deploying patches is relatively harder than instrumenting the binary.

4. Post-deployment adaptive instrumentation, on the other hand, facilitates easy deployment, while

relying on selective instrumentation used by adaptivity to guarantee low overheads. It cannot take advantage of post-instrumentation compiler optimizations.

The evaluations in this chapter, except the performance measurements in section 4.4.4, hold true irrespective of whether selective instrumentation is done before or after deployment.

Security Implications

Our proposal on adaptive bug isolation opens one more channel of communication between the software vendor and the user. This raises concerns about a malicious third-party forcing instrumentation to collect sensitive user data. We can use existing infrastructure to alleviate this concern. Most consumer software has the ability to securely and automatically check for updates, and apply them.

Another security concern is the trustworthiness of the binary instrumentor. The binary instrumentor is trusted by the end-user to perform updates to the application. In the system we envision, the binary instrumentor is deployed with the same privileges and protection as the update manager. In this situation, either the instrumentor is safe from tampering, or is not allowed to instrument the application if a compromise is detected.

Deviations in Usage Patterns

One underlying assumption of adaptive bug isolation is with regards to usage patterns. The users' interactions with the software, and consequently the symptoms of failure and best predictors of failure, are assumed to be stable over time. Adaptive bug isolation adds instrumentation in locations that are most likely to be useful for the failed runs seen so far. This set could be less useful if the usage pattern changes, and the program now fails under different circumstances. This situation is alleviated by the idea developed in section 4.4.3 to handle multiple bugs. However, stable usage patterns for long periods is still desirable when performing the iterative refinement of instrumentation predicates. The speed at which the reports are collected also factors into this question. Our hypothesis is that deployment of our technique in a large user community will yield faster accumulation of runtime feedback, as well as provide pockets of the user

community with stable behavior. The small overheads we demonstrate in section 4.4.4 can encourage the participation of such large user communities.

4.2 Binary Instrumentation

This section evaluates the option of fixed, post-deployment instrumentation in table 4.1. We use the Dyninst [BH00] instrumentation framework, which allows many optimizations that are difficult, and in some cases impossible, to achieve using other tools for binary instrumentation. This section discusses those optimizations in detail. Unfortunately, even with these optimizations, overheads are too large for deployment to end users (see section 4.2.4). This motivates using an adaptive approach, discussed in section 4.3, to achieve truly lightweight monitoring.

4.2.1 Basic Instrumentation and Reporting

Dyninst explicitly separates the program being instrumented from the program directing the instrumentation. We term the former the *target* and the latter the *instrumentor*. The instrumentor launches the target in a separate address space and leaves it in a suspended state. Dyninst reconstructs the target's global call graph and per-function control-flow graphs. From this, we identify branches and function call return points where instrumentation will be inserted. Each instrumentation site requires two (for branches) or three (for function returns) global counters. For reasons we discuss below, these counters are stored in a shared memory segment mapped into the address spaces of both the instrumentor and the target. The instrumentor generates code to increment the predicate counters and inserts this code into the target. The target is then released from the suspended state and is allowed to run to completion.

On termination, predicate counters must be read and stored for later analysis. One approach would be to register a process-exit callback in the target, such as via ANSI C's `atexit`. However, an exiting target may already have corrupted its heap, run out of memory, or be in an otherwise unstable state. We cannot assume that it is still capable of sending counter values across a network or saving them to a file. Instead, our instrumentor detects when the target has exited, and then reads the predicate counters directly from

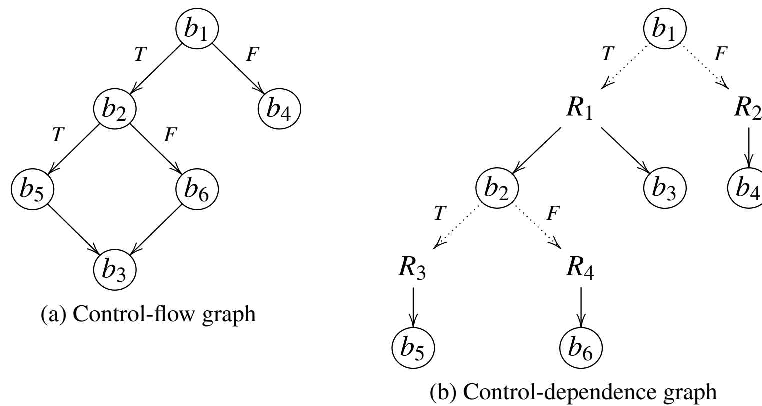


Figure 4.3: Example graphs for static removal of branch predicates

the shared memory segment. The instrumentor also records the target application's exit status at this time, for use when labeling crashes as failed runs.

4.2.2 Static Removal of Instrumentation

Branch predicate counts are equivalent to the edge profiles of an execution. We use static program structure to avoid redundant operations using approaches similar to those of Ball and Larus [BL96] or Tikir and Hollingsworth [TH05]. However CBI collects more than just edge profiles. Coverage of any predicate, such as a function return predicate, implies coverage of the basic block in which that predicate is defined. This in turn implies coverage of the edges leading to that block from all of its control-dependence ancestors.

Figure 4.3 shows an example control-flow graph (CFG) and the corresponding control-dependence graph (CDG). The true and false edges of branches are labeled T and F respectively. The additional R_i nodes in the CDG are *region nodes* [Fer+87] that group nodes with identical control conditions. For example, nodes b_2 and b_3 execute if and only if the condition at b_1 is true.

Ordinarily, a branch instrumentation site at b_1 would place one predicate counter along the edge from b_1 to b_2 and another along the edge from b_1 to b_4. If there is already any instrumentation site s at b_2 or b_3, then the branch predicate count along the edge $b_1 \rightarrow b_2$ must equal the sum of the counts of the predicates at s. Thus, the $b_1 \rightarrow b_2$ edge need not be instrumented and its missing edge profile can be computed offline. In general, we define a branch predicate as *redundant* if its count can be inferred offline by a post-mortem

analysis of the execution profile. Consider a potential branch predicate p along the edge from some block u to a successor block v. Predicate p is redundant if u dominates v and at least one instrumentation site s is defined in v or any other basic block that is *control-equivalent* to v. The first condition ensures that the entire *control-dependence region* corresponding to v is executed if and only if the edge from u to v is traversed. The second condition ensures that every execution of the control-dependence region containing v will be reflected in one of the predicates from site s. Under these conditions, instrumentation for branch predicate p can be omitted. The count for p can be derived offline by summing up the counters for the predicates at site s.

Control-flow graphs and derived representations rarely reflect the possibility that buggy programs may terminate abruptly. Doing so would create additional edges from nearly every node to the global program-exit node. Such edges would confound our ability to assume that site s is reached every time the edge $u \to v$ is crossed, because it is possible to cross $u \to v$ but crash before reaching s. Fortunately, this situation can be recognized and corrected by inspecting the stack of a failed run. For each stack frame that shows execution in the control-dependence region corresponding to v but before reaching s, the sum of counts at s should be incremented by one to derive the actual count of $u \to v$ edge crossings.

4.2.3 Binarization and Dynamic Removal

While some statistical debugging models use exact values of predicate counts [And+07; JS07; Liu+05; Zhe+06], others require only *binarized data*: they consider only whether a predicate was true at least once, but make no further distinctions among nonzero counts [JH05; Lib+05]. If a binarized model is to be used, then predicate "counters" are merely flags. Instrumentation code can just store a 1, which takes one memory operation, instead of incrementing a counter, which takes one arithmetic and two memory operations.

Furthermore, when using binarized data, there is no benefit from additional observations of a predicate that has already been observed true once. Therefore, a predicate's instrumentation code may be removed from the target once it has triggered [CE04; Mis+05; TH05]. Dynamic instrumentation removal is especially well suited for branch instrumentation, as each branch predicate adds code on a distinct edge and therefore

each branch predicate can be removed independently.

4.2.4 Performance Impact

Instrumentation must have extremely low overhead if we are to collect feedback data from members of the general public. Experiments with a small, CPU-intensive benchmark show that naïve binary instrumentation does not achieve this goal.

We use the SPEC `099.go` benchmark, compiled with gcc 4.1.2, instrumented using a beta version of the Dyninst 6.0 release and run on an otherwise idle dual-core 3.2 GHz Pentium CPU. We use one small (2stone9) and two large (5stone21 and 9stone21) benchmark workloads. All measurements reported are averages across five repeated trials. Execution time excludes instrumentor start-up costs and reflects only time spent running the instrumented code. Start-up costs can be amortized over several runs using the new binary rewriting feature in Dyninst. We instrument all branches and function returns in the main executable but not in shared libraries.

The unmodified `go` executable completes the small workload in 0.4 seconds, and the two large workloads in 21.7 and 21.6 seconds respectively. Naïve Dyninst instrumentation slows execution by a factor of 5.8 times for the small workload and 5.5 times for the large workloads. Adding static branch instrumentation removal, binarized counts, and dynamic branch instrumentation removal increases this relative slowdown to 8.9 for the small workload, but shrinks it to 1.8 for the large workloads: the benchmark's small code footprint means that dynamic branch instrumentation removal is more beneficial for longer-running tasks. While 1.8 is better than 5.5, this is still too slow. Users will not accept a 1.8× slowdown in daily use.

We also consider three non-Dyninst-based approaches: *Pin*, *Valgrind* and *sampler-cc*. Pin and Valgrind are dynamic binary instrumentors that use *just-in-time* (JIT) disassembly and recompilation, as contrasted with Dyninst's code-patching approach. Our custom Pin instrumentor built using Pin version 2.6 has slowdowns between 4.3 and 5.0. Our custom Valgrind instrumentor built using Valgrind version 3.2.1 performs similarly to lightly-optimized Dyninst, with slowdowns between 7.0 and 10.2. However, many of the more aggressive optimization strategies would not be practical to apply under a JIT execution model. Moreover, just-in-time code patching maintains a code cache to hold instrumented basic blocks. This

code cache imposes a baseline overhead to load and execute instructions even when no instrumentation is performed. This limit cannot be improved with any static or dynamic optimization. For Pin, this overhead comes to about 1.9, which is higher than highly optimized Dyninst instrumentation. For this reason, we evaluate our adaptive techniques in the next section using Dyninst. As opposed to our binary instrumentor, sampling based instrumentation at the sparsest sampling rate of $1/\text{UINT_MAX}$ imposed relative slowdowns of 1.4 across all workloads. While source-level information based on sampling is the fastest among the options considered here, we are not willing to give up the benefits of dynamic instrumentation while simultaneously imposing a 40% slowdown on end users. The adaptive techniques described in the next section dramatically reduce instrumentation overheads while simultaneously avoiding the drawbacks of static instrumentation.

4.3 Adaptive Instrumentation

The goal of statistical debugging algorithms is to find predicates that are correlated with failure. Existing statistical debugging techniques attempt to find the tiny fraction of predicates that are predictive of failure. However, they assume that instrumentation sites are selected once and remain fixed thereafter. This section describes an adaptation-based approach that eliminates fixed monitoring plans. Instead, sites are speculatively added to the instrumentation plan if it appears that they may be good predictors of failure and are removed from the monitoring plan once their ability (or inability) of failure prediction has been assessed. Our algorithm exploits the principle of locality: if a predicate is highly predictive of failure, then predicates in its vicinity are potentially good failure predictors as well. Our experience with statistical analysis using non-adaptive instrumentation found that the principle of locality holds in some, but not all cases. The essence of our technique is to adaptively adjust the instrumentation plan by locating a predicate that is highly predictive of failure and extending the plan to include nearby sites.

Procedure 1 defines this iterative algorithm. It is parameterized by four sub-procedures (*GetInitialSet*, *WaitForSufficientData*, *score*, and *Vicinity*) that are described more fully later. *monitored* is the set of sites for which feedback information is available from previous iterations. *explored* is the set of branch predicates that have received the highest score in previous iterations; these are predicates whose nearby

1: $monitored = \emptyset$
2: $explored = \emptyset$
3: $plan = GetInitialSet()$
4: **while** debugging **do**
5: Instrument and monitor sites in *plan*
6: $WaitForSufficientData()$
7: $monitored = monitored \cup plan$
8: *best* = branch predicate with highest *score* in $monitored \setminus explored$
9: $explored = explored \cup \{best\}$
10: $plan = Vicinity(best) \setminus monitored$
11: **end while**

Procedure 1: Pseudo code for Adaptive Analysis

vicinity has already been explored. *plan* is the set of sites that are being monitored during the current iteration. *best* is the branch predicate that receives the highest score in the current iteration. At startup, the analysis chooses the set of sites to be monitored by calling *GetInitialSet*. The set of sites is monitored until the function *WaitForSufficientData* returns, indicating that enough feedback has been collected for meaningful analysis to be applied. Using this feedback, and feedback from earlier phases (if any), the best branch predicate that was not already explored is identified. The plan for the next iteration is to monitor previously unmonitored sites in the vicinity of this best predicate as defined by the function *Vicinity*. The sets *monitored* and *explored* are updated during each phase. Note that we are not monotonically adding instrumentation to the program. The selection of the new plan done in line 10 of procedure 1, in addition to adding sites returned by *Vicinity*, stops instrumentation of predicates already monitored.

During any iteration, the programmer can view analysis results for all sites monitored previously. The programmer can choose to continue the adaptive search, or may stop the process if he recognizes and can fix the root cause of the bug. Thus, termination of the "**while** debugging" loop is a human decision. Procedure 1 describes an automated search that can proceed without human intervention, but it is assumed that human programmers are observing, learning from, and occasionally intervening in the process.

Our choice of algorithms for the sub-procedures *GetInitialSet* and *Vicinity* determines whether the analysis in procedure 1 is a forward (section 4.3.1) or backward (section 4.3.2) analysis. The *score* function assigns numeric values to every predicate; a higher value is assigned to a predicate that is a better predictor of failure. We describe some candidate scoring functions in section 4.3.3 and experimentally evaluate

them in section 4.4. Some scoring heuristics developed for non-adaptive instrumentation are designed to compute the inherent bug-predictivity of a predicate and prevent the bug predictivity of nearby predicates from skewing the measure for this score. Such metrics are less suitable for our adaptive algorithm, as they subvert the locality principle on which it relies. *WaitForSufficientData* assesses whether sufficient data has been collected. While not the main focus of this work, we briefly discuss this issue in section 4.3.4. Section 4.3.5 mentions some alternative design choices.

4.3.1 Forward Analysis of the Program

The general pattern in forward-adaptive bug isolation is to start at the beginning of the program and iteratively work forward toward the root causes of bugs. Consider the control-flow graph in fig. 4.3a. Suppose the branch predicate associated with the true result of the condition at b_1 is found as the best predicate according to the *score* function. This means that whenever the edge $b_1 \rightarrow b_2$ is traversed, the program is likely to fail. This indicates that there might be some bug in the basic blocks b_2, b_3, b_5 and b_6 and predicates in these blocks may be even better at predicting the bug. However, we do not have enough evidence to believe that b_5 and b_6 have good failure predictors. It could be the case that the bug is in b_5 and hence none of the predictors in b_6 predict failure. It is also possible that the bug is in b_6 and the predicates in b_5 are not relevant. Which of the two cases is true will be known when we have information about the branch site at b_2. We can defer monitoring sites in b_5 and b_6 until we have that information.

On the other hand, we have enough reason to believe that good predictors will be found in b_2 and b_3 because the collected data shows that these blocks are executed in many failed runs. In general, the choice of b_2 and b_3 translates to choosing the children in the control-dependence graph (CDG). Thus, if the *best* predicate is associated with the branch at basic block b being true (respectively, false), then we are interested in the basic blocks that are control dependent on b with the true (respectively, false) control condition. Therefore *Vicinity*$(best)$ returns the sites in these basic blocks. Function calls are handled automatically by using an interprocedural control-dependence graph. To fit with the notion of searching forward in the control-dependence graph, *GetInitialSet* returns the sites in basic blocks control dependent on the entry node of the CDG.

4.3.2 Backward Analysis of the Program

If a program fails by crashing, then a stack trace of the program when it crashed may be available. Based on the folk wisdom that the bug is likely to be somewhere near the point of the crash, exploring the predicates near the crash point may find good bug predictors faster than a forward analysis. To illustrate backward analysis, once again consider the control-flow graph fragment in fig. 4.3a. Suppose the program crashes in block b_3. Now consider the branch site at b_1, which is b_3's control ancestor. This site is the last point where execution of b_3 could have been skipped and hence the crash averted. So, b_1 is a good candidate bug predictor and we monitor it and measure its score.

Suppose the programmer looks at the new feedback and has no idea why a predicate at b_1 could be causing the crash. There are two possible reasons:

case I: The bug may actually be in basic block b_2, b_5, or b_6. The predicate at b_1 may have a high score simply because it governs execution of these blocks.

case II: The branch predicate at b_1 may have a high score because the problem happens before the program reaches b_1. Thus, the program will fail irrespective of the outcome of this branch.

In case I, predicates in b_2, b_5 and b_6 are potential bug predictors. Using the same reasoning used during forward analysis, we only monitor b_2 and delay monitoring of b_5 and b_6 until there is information about the branch predicates in b_2. In case II, we could explore further backwards in the CDG by considering b_1's control ancestor (assuming that fig. 4.3a shows just a fragment of a larger program). Since there is no way to decide whether the root cause is before the execution of b_1 or after it, we take a conservative approach and include sites suggested by both cases I and II in the monitoring plan for the next iteration.

To summarize, for backward analysis, *GetInitialSet* returns the branch site in the control ancestor of each basic block in which a crash occurs. *Vicinity*(*best*) returns sites in the control ancestors and control children having the appropriate control condition of the basic block in which the predicate *best* is defined.

4.3.3 Scoring Heuristics

In this section, we consider possible definitions for *score* as used in procedure 1. All possibilities considered are heuristics in that one could contrive situations in which they perform badly. Our goal is to identify scoring heuristics that perform well on a variety of programs in realistic situations.

In the sites-and-predicates model used by CBI, the data collected for a predicate p is aggregated into four values: $S(p)$ and $F(p)$ are, respectively, the number of successful and failed runs in which p was observed to be true at least once. $S(p \text{ obs})$ and $F(p \text{ obs})$ are, respectively, the number of successful and failed runs in which p was observed at least once regardless of whether it was true or not. Besides these four values, there is another global value: *NumF*, the total number of runs that were labeled as failures. All heuristics described in this section are computed using these values.

Failure Counts

The first heuristic scores a predicate p according to the number of failed runs in which p was observed to be true: $\mathit{FailCount}(p) \equiv F(p)$. Any region of code that is executed during many failed runs is potentially buggy. $\mathit{FailCount}(p)$ may not be a good measure of failure predictability because it does not distinguish between two predicates that are true in different numbers of successes but the same number of failures. However, a predicate seen in many successful and failed runs may capture some property of the program other than its outcome, such as differing usage scenarios [Zhe+06].

Importance

The *Importance* score was introduced in section 2.3.1 as a good measure of failure predictivity. We can also consider it as a candidate-heuristic for adaptive scoring. For convenience, the definition of *Importance*

is as follows:

$$\begin{aligned}
\textit{Sensitivity}(p) &\equiv \frac{\log(F(p))}{\log(\textit{NumF})} \\
\textit{Increase}(p) &\equiv \frac{F(p)}{S(p)+F(p)} - \frac{F(p \text{ obs})}{S(p \text{ obs})+F(p \text{ obs})} \\
\textit{Importance}(p) &\equiv \frac{2}{\frac{1}{\textit{Increase}(p)} + \frac{1}{\textit{Sensitivity}(p)}}
\end{aligned}$$

Maximum Importance

During initial experiments, we found that *Importance* as a scoring heuristic often leads to sub-optimal adaptation decisions. For example, consider a branch condition that is always true and the associated branch predicate p. Since the branch is always taken, $F(p \text{ obs}) = F(p)$ and $S(p \text{ obs}) = S(p)$. Thus $\textit{Increase}(p)$ and consequently $\textit{Importance}(p)$ are both 0. Even if $F(p)$ is very large, a branch predicate p' with marginally positive values for $\textit{Increase}(p')$ but very low $F(p')$ will be given preference over p. This is not a problem with the *Importance* heuristic because, as mentioned earlier, there can be situations where the heuristics make choices that go against the principle of locality. If the iterative ranking and elimination algorithm described in section 2.3.2 is used, where the goal is to find predicates with high *Importance* scores, we can construct a heuristic that maximizes the *Importance* score of predicates in $\textit{Vicinity}(p)$ rather than the *Importance* of p itself.

Predicates in $\textit{Vicinity}(p)$ have not already been monitored, so we cannot predict the exact maximum score among predicates in that set. Instead, we compute an upper bound by considering a hypothetical predicate h that has the best possible score. Such a predicate must be true in all the failed runs in which it is observed and false in all the successful runs in which it is observed, i.e. $F(h) = F(h \text{ obs})$ and $S(h) = 0$. When a forward analysis is used, h will appear in a basic block that is control dependent on the edge associated with p and hence h will be observed only when p is true, so $F(h \text{ obs}) = F(p)$ and

$S(h \text{ obs}) = S(p)$. Thus,

$$\begin{aligned}
\mathit{Increase}(h) &= \frac{F(h)}{S(h)+F(h)} - \frac{F(h \text{ obs})}{S(h \text{ obs})+F(h \text{ obs})} \\
&= 1 - \frac{F(p)}{S(p)+F(p)}
\end{aligned}$$

We set $\mathit{MaxImportance}(p) \equiv \mathit{Importance}(h)$ to favor predicates that have the potential to reveal new predicates with high *Importance*.

Student's t-Test

The next heuristic, $\mathit{TTest}(p)$, uses a statistical test called the Student's t-test [Lev69]. During a forward analysis, if p is true in a larger percentage of failures than successes, then the predicates in $\mathit{Vicinity}(p)$ are also observed in a larger percentage of failures than successes and the heuristic should give preference to p. On the other hand, if p is true in a larger percentage of successes than failures, then the predicates in $\mathit{Vicinity}(p)$ should have lower preference. There are two sets of data for each predicate p: the observations of p in successful and failed runs. The null hypothesis is as follows:

> H_0: the observed behavior of p in successful and failed runs are indistinguishable

The t-test uses the mean, standard deviation and the size of the two samples to assign a numeric confidence in the range $[0, 1]$ with which the null hypothesis can be rejected. While not the usual interpretation of the t-statistic, we use this confidence to characterize whether the observed behavior of p in successful and failed runs differ significantly. The $\mathit{TTest}(p)$ heuristic computes the t-test confidence measure c and assigns a score of c to p if it is seen in more failures than successes. If p is seen in more successes than failures, a score of $-c$ is assigned to p. Since the information about p does not impose any useful restrictions on the coverage of the ancestor of the node associated with p in the CDG, neither *MaxImportance* nor *TTest* has a sensible interpretation for backward analysis.

Student's t-test is a *parametric* test and assumes that the samples are normally distributed. We also evaluated a *non-parametric* test, the Mann–Whitney U-test [MW47], which is less powerful but does not

assume normal distribution. The U-test never performed better than the t-test indicating that the normality assumption is acceptable. Therefore, we give the U-test no further consideration.

Other Heuristics

To evaluate the usefulness of our techniques, we also define three other heuristics. To assess whether search heuristics are useful at all, we consider *BFS*, a naïve breadth-first search on the CDG. *Random* is a straw man heuristic that selects a "best" predicate at random on each iteration. Lastly, an *Oracle* heuristic helps to measure how well any search heuristic could possibly do; we discuss *Oracle* further in section 4.4.2.

4.3.4 Waiting for Sufficient Data

Most statistical analyses have the ability to associate a confidence value with their output. A typical confidence measure would be a probability, between 0 and 1, that an observed trend is genuine rather than merely coincidental. Given such information, we can wait until the analysis is able to compute its output with a sufficiently high confidence (for example, with probability greater than 0.95). In general, this decision involves a compromise between the speed and accuracy of debugging. Collecting more reports improves accuracy but may take longer to produce interesting results, while collecting fewer reports has opposite trade-offs. Moreover, user behavior might change across iterations and more reports will be needed to accommodate such instability. As mentioned earlier, this issue is not the main focus of this work. Section 4.4 explains a simple approach that we use for our experimental evaluation.

Note that the definitions of *GetInitialSet* and *Vicinity*, as given in either section 4.3.1 or section 4.3.2, have an important completeness property. With either approach, starting with *GetInitialSet* and repeatedly expanding the search using *Vicinity* will eventually instrument every site that is reachable from the program's entry point. This property is important because the principle of locality is not a guarantee: a good predicate may appear in code where other predicates have low scores. The completeness property ensures that adaptive bug isolation can recover from wrong turns. If the "**while** debugging" is allowed to run until all sites have been instrumented, provide the same information as exhaustive (non-adaptive)

instrumentation; it may just take longer to get there.

4.3.5 Design Alternatives

As proposed here, adaptive instrumentation is a heuristic search through the control-dependence graph, and the principle of locality is assumed to apply to predicates that are close in this graph. However, the effects of bad code can also propagate through data rather than through control flow. Thus, it may be desirable to consider data-dependence relations as well. This can be done by using the *program-dependence graph* (PDG) [HR92] instead of the control-dependence graph. Balakrishnan et al. [Bal+05] have demonstrated PDG construction for unannotated binaries.

Instead of constructing an interprocedural CDG, one could initially treat calls as opaque and only instrument callee bodies if the results they return are highly predictive of failure. However, this incorrectly assumes that called functions are pure, with no effects other than the values they return. This clearly is not true for C. Thus, a completely modular, function-by-function search is not appropriate in the general case.

Procedure 1 describes an automated search that can proceed without human intervention. But the general framework is flexible and can be manually overridden if and when needed. For example, the programmers can override *GetInitialSet* to directly debug modules that are known to be buggy from prior experience or in-house testing. Similarly, experienced programmers can look at already-monitored sites and specify their own plan based on domain knowledge, whether to test a hypothesis or simply chase down a hunch. The set of monitored sites can be actively modified to balance aggressive exploration against user-tolerable overhead.

4.4 Evaluation

Prior work [And+07; Chi+09; JS07; Lib+03; Lib+05; Liu+05; Zhe+06] has shown qualitatively and quantitatively that statistical debugging is effective. In this section, we evaluate the main contribution of this chapter, which is the use of adaptive binary instrumentation to further reduce performance overheads. As mentioned earlier, our technique is geared towards retaining the fault localization ability of existing statistical analyses, while also achieving low overheads, and avoiding instrumentation of bug-free code.

Table 4.2: Programs used for experimental evaluation

Program	Variants	LOC	Sites	Test Cases
`bash`	1	59,846	17,996	1,061
`bc`	1	14,288	1,799	10,000
`ccrypt`	1	5,276	757	10,000
`exif`	1	10,588	2,631	10,000
`flex`	47	14,705	2,538	567
`gcc`	1	222,196	56,850	892
`grep`	17	14,659	2,666	809
`gzip`	9	7,266	1,406	217
Siemens	7 to 41	173 to 563	94 to 184	1,052 to 5,542
`space`	38	9,126	1,673	13,585

Our evaluation uses the faulty programs of the Siemens suite [Hut+94]; the `bash`, `flex`, `grep`, `gzip` and `space` bug benchmarks [Rot+06]; and `gcc` 2.95.3 [Fre]. We also evaluated `bc` 1.06, `ccrypt` 1.2 and `exif` 0.6.9, each of which has known fatal bugs [Lib+05]. Some test subjects have multiple variants, each exhibiting a different bug. Table 4.2 lists our test programs, the number of variants, their size in lines of code and number of instrumentation sites, and the size of the test suite used. All test programs except the Siemens programs are realistic applications with several thousand lines of code (KLOC). The largest applications are `bash` and `gcc`, with greater than 50 KLOC.

The scalar-pairs scheme is not supported by our binary instrumentor. The main roadblock in adding support for this scheme is the difficulty in reliably identifying the whereabouts of each variable from just the binary. Techniques for deeper analysis of executables [BR07] might help in this task. However, predictors belonging to the branches and returns schemes are the top-ranked predicates for 57% of our test subjects. Even among the remaining 43% of the subjects that had a top-ranked scalar-pairs predicate, predicates belonging to the other schemes have good scores. In section 3.4.2, we manually evaluated the bug-relevance of the top-ranked predicates. Among the 36 subjects where simple predicates were found relevant for bug-isolation, 22 (i.e. 61%) belonged to the branches and returns schemes. Thus, omitting the scalars-pairs scheme does not severely impact the usefulness of predicates found using binary instrumentation.

Bugs in some test subjects cause incorrect output rather than crashes. Like the experiments in chapter 3,

a test case is labeled as a success or failure by comparing the output of the buggy program to that of a bug-free reference version. Our statistical analyses are applicable irrespective of the labeling strategy used. For the purposes of backward analysis, the failure point is defined as the statement in the program that prints the first incorrect byte in the output. We find this location by tracing program output and associating each output byte with the code that printed it [Hor+10]. For each failed test case, the traces from executions of faulty and reference versions are compared to find the failure point. Since the output tracer does not support `bash`, we do not perform backward analysis for `bash`.

The *Importance* score is undefined if a variant fails in zero or one test case. Such variants are discarded and not included in table 4.2. For the remainder, we run the adaptive analysis given in procedure 1. To mimic a real deployment in which no two runs are exactly alike, we partition tests into random subsets of 500 cases each and cycle through these for successive iterations. When using the *Random* heuristic, all measures are averages across five trials.

4.4.1 Comparison of Heuristics

The intent of each heuristic is to guide adaptive analysis toward high-scoring predicates. To evaluate effectiveness of the heuristics, we plot the score of the top-scoring predicate found versus the total number of sites monitored so far. Figure 4.4 shows these plots for large applications (`bash` and `gcc`), medium-sized applications (`bc`, ccrypt, exif, grep, space) and the small applications in the Siemens test suite. The heuristics deviated from the normal pattern of the medium-sized applications for `gzip` and `flex`. The plot for `gzip` is shown in fig. 4.4d.

Note that *Vicinity*(*best*) in procedure 1 may return multiple sites. When this occurs, flat horizontal segments appear in the plots of fig. 4.4, reflecting a larger-than-unit jump in the number of sites monitored. This is particularly common with *BFS*, which can fan out quickly in each iteration.

For the medium-sized applications (fig. 4.4b), all heuristics start by finding essentially the same top score before diverging at 100 sites. The divergence at 100 sites may look small in the plot, but it is significant considering the large range covered by the *x* axis. *TTest* performs best followed by *FailCount*. *FailCount* lags *Importance* in the early phases because it cannot distinguish predicates seen only in

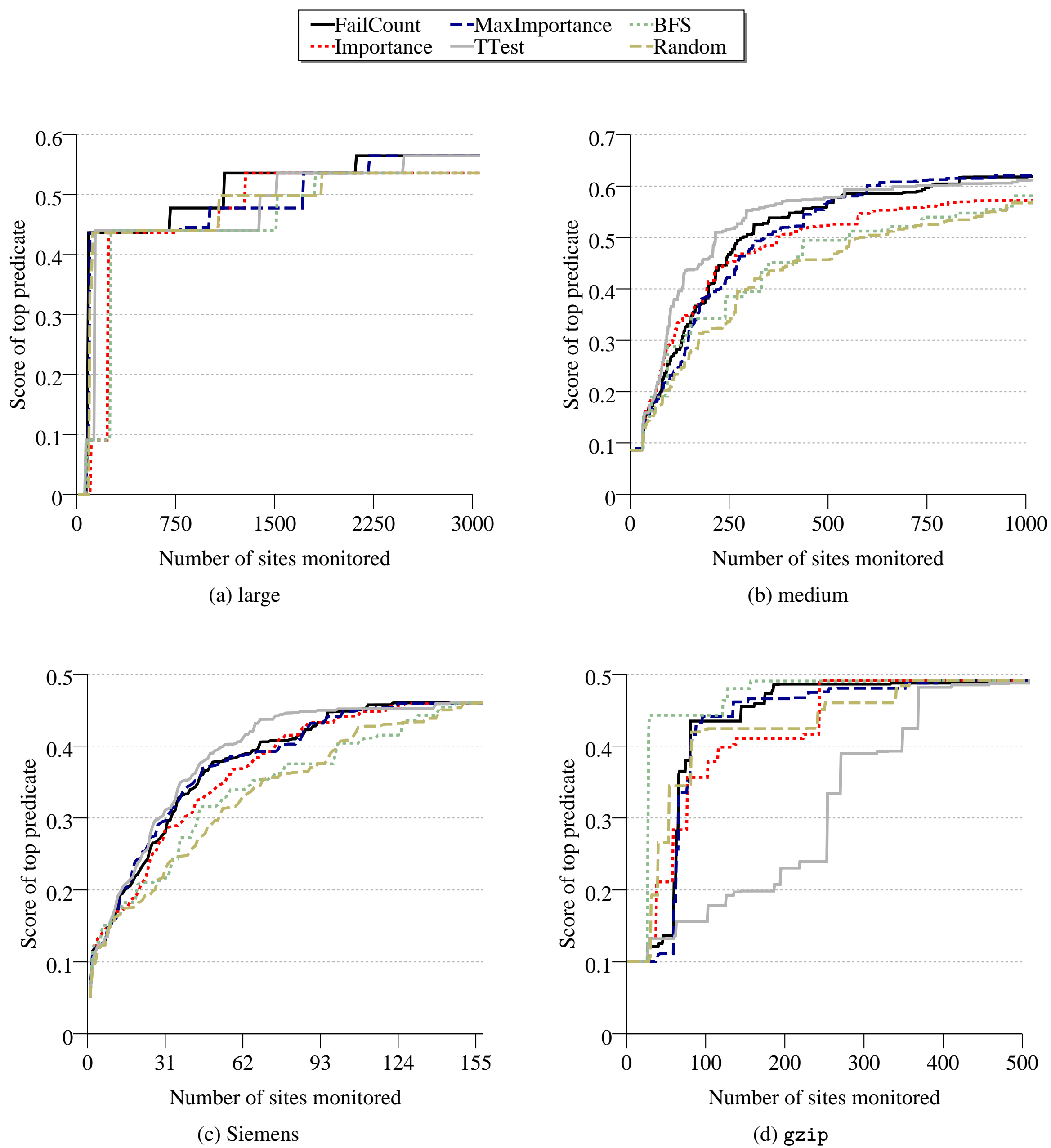


Figure 4.4: Adaptation speed for various heuristics using forward analysis

failed runs from those seen always. The pathological cases for *Importance* that motivated the design of *MaxImportance* appear to be fairly common in these applications, as indicated by the fairly good performance of *MaxImportance*. Wide fan-out leads *BFS* to instrument many uninteresting sites on its way to a good predicate. All of these outperform the *Random* straw man. The heuristics behave similarly for the Siemens programs: *BFS* and *Random* are significantly worse than the other heuristics; *TTest* is the best or close to the best; the others are close to *TTest* for some cases and close to *BFS* in the rest. For the large applications `bash` and `gcc` (fig. 4.4a), *FailCount* is the best heuristic, followed by *Importance* and *TTest*. For programs `gzip` and `flex`, the heuristics deviate from the general pattern observed above. In the plot for `gzip` in fig. 4.4d, *BFS* and other heuristics significantly outperform *TTest*. Manual inspection shows that the top predictors are near the top of the CDG (usually two or three levels deep). Thus, *BFS* and *Random* are better at finding them early, while *TTest* gets sidetracked by an initial wrong choice. These are ultimately heuristics, and therefore can occasionally perform sub-optimally. `flex` exhibits behavior similar to `gzip`.

Figure 4.5 compares backward analysis performed using the *Importance* heuristic and forward analysis using the *TTest* heuristic. (Backward *TTest* is meaningless.) The subplots show the increase in the score of the top-scoring predicate found versus the total number of sites monitored for the large, medium-sized, and the small Siemens applications.

We do not perform backward analysis for `bash`, because the output tracer used to find the crashing location does not support it. Figure 4.5a compares forward and backward analysis for `gcc`, the other large application. Backward analysis does not find the best predictor even after 1,500 iterations. The difference is entirely incidental, and has to do with the presence of boiler-plate code in `gcc` that parses the input source file. The predicates in this region of code are not predictive of failure because they are executed in all runs. The crash location and the top predictor are separated by this boiler-plate code. Backward analysis is unable to make the right choice when exploring predicates in this region, and hence is unable to find the top predicate. Forward analysis, on the other hand, is able to find the top predictor without having to deal with this problem.

For the medium-sized applications, backward analysis begins finding good predictors earlier. This

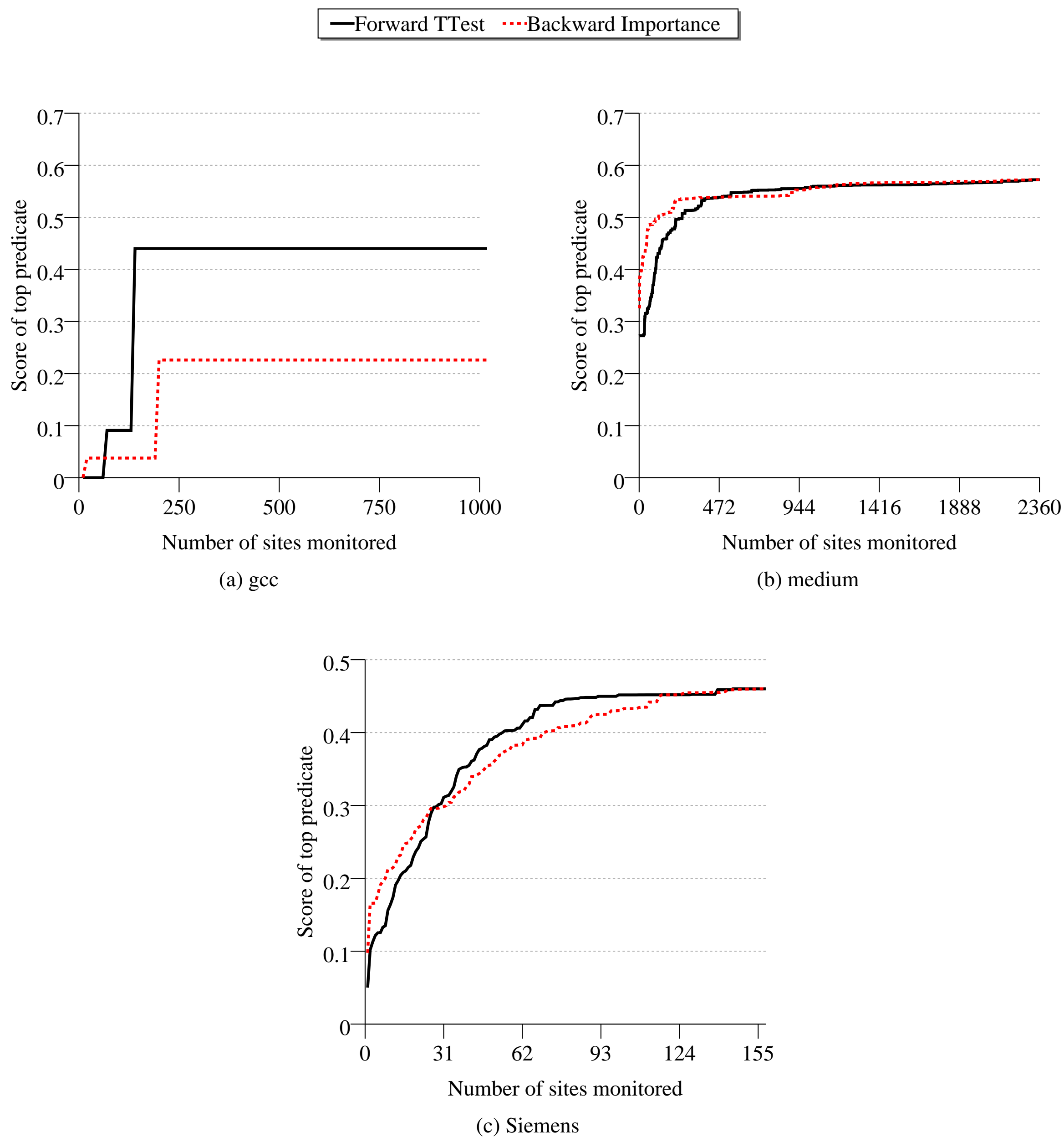


Figure 4.5: Adaptation speed for various programs using forward and backward analysis

affirms the folk wisdom that many bugs are close to their points of failure. After about 200 sites, exploration does not improve the scores significantly. The increase in the score of the top predicate around 950 sites indicates that for some variants, the highest-scoring predicate is not found until later iterations. This suggests that the principle of locality does not always apply; high-scoring predicates occasionally appear in the same locality as low-scoring predicates. For `bc` and `exif`, which have crashing bugs, the top predictor is very close to the point of failure and is found rapidly by backward analysis, after exploring fewer than 100 sites. For `gzip`, backward analysis is better, finding the top predicate after exploring about 200 fewer sites than forward analysis. For `space`, `ccrypt`, and `flex`, backward analysis performs better initially, but forward analysis wins in the long run.

For the Siemens programs, backward analysis starts marginally better than forward, but is soon overtaken. This does not mean that backward analysis is less useful: the Siemens programs are small (about 500 lines of code) and most output is printed at basic blocks at or close to the top of the CDG. Thus, backward exploration stops fairly early and only a forward analysis is performed thereafter. To conclude, backward analysis is better initially but forward analysis is just as effective in the long run.

4.4.2 Instrumentation Selectivity

In the previous section, we evaluated the first goal of adaptive bug isolation, which is to prioritize instrumentation of good failure predictors. We used the variation in *Importance* scores of predicates over time for this evaluation. Our second goal is to avoid instrumentation of predicates that are not failure predictive. This can be evaluated by using the fraction of predicates that were never instrumented during the debugging activity. This requires identification of the point in time when debugging finishes, which is not precisely defined. It depends on the skill and motivation of the developer, in addition to the quality of the failure predictors found by our technique. The human element is hard to capture. Instead, we consider the top predictor found by the iterative bug isolation algorithm (section 2.3) as the reference. We consider debugging to have finished when our adaptive approach finds this top predictor. We make the simplifying assumption that a lazy programmer who looks at only the first predicate in the bug-isolation output will find adaptive bug isolation just as useful as non-adaptive statistical debugging.

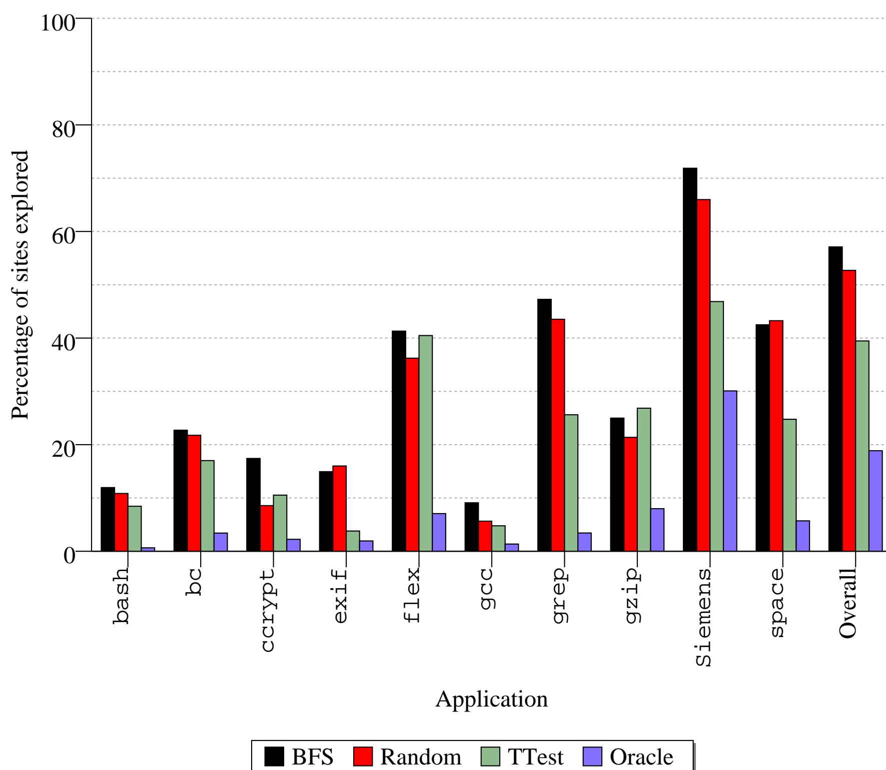


Figure 4.6: Mean number of sites to find top-ranked predictor

To gauge the progress towards our second goal, we measure the total instrumentation effort required for the adaptive process to discover the same top-ranked bug predictor as a traditional, non-adaptive method. For each variant, we note the top bug predictor, and then count sites explored before an adaptive analysis identifies this same predicate. We compare *TTest* with *BFS* and *Random* to test whether a carefully-selected heuristic can help in this task. We also consider an *Oracle* heuristic that has perfect knowledge of program behavior. It always selects the branch predicate that reaches the target predicate by monitoring the fewest instrumentation sites.

Figure 4.6 plots the percentage of sites explored for each program averaged across all variants. *BFS* and *Random* explore about 60% of sites before finding the top predicate. *TTest* explores just 40%. *Oracle* suggests that there is room for improvement but also establishes a lower bound of about 20% for any adaptive search that crosses CDG edges one at a time. Adaptive analysis performs very well in `bash`, `bc`,

Table 4.3: Mean number of sites instrumented per iteration

Program	FailCount	Importance	MaxImp	TTest	Random	BFS	BwImp
`bash`	3.9	3.4	3.2	2.8	2.6	179.2	-
`bc`	3.6	2.9	4.3	4.8	2.8	60.9	163.0
`ccrypt`	3.3	2.4	5.3	2.6	3.6	24.8	4.5
`exif`	1.9	2.4	1.9	3.0	2.3	24.6	2.7
`flex`	46.7	45.4	51.7	44.1	43.8	289.6	137.5
`gcc`	3.8	3.2	3.7	4.2	3.8	517.2	12.0
`grep`	5.8	6.1	6.3	5.9	5.1	110.4	9.7
`gzip`	5.5	10.1	6.5	9.8	6.5	105.3	10.6
Siemens	3.2	3.1	3.1	3.2	2.9	9.9	3.5
`space`	3.3	3.6	3.1	3.9	2.6	56.1	18.3

Table 4.4: Mean number of iterations for each program

Program	FailCount	Importance	MaxImp	TTest	Random	BFS	BwImp
`bash`	181.0	314.0	316.0	545.0	766.4	12.0	-
`bc`	94.0	156.0	73.0	66.0	145.2	7.0	1.0
`ccrypt`	11.0	79.0	9.0	29.0	16.8	5.0	59.0
`exif`	73.0	99.0	70.0	33.0	186.6	16.0	22.0
`flex`	176.3	242.5	67.6	311.4	240.8	3.1	114.7
`gcc`	618.0	1227.0	662.0	645.0	863.3	10.0	1500.0
`grep`	180.2	388.2	107.5	299.2	546.3	10.6	190.2
`gzip`	57.7	42.6	50.8	127.8	84.3	3.1	87.6
Siemens	19.7	25.6	20.7	18.6	30.6	7.7	21.2
`space`	148.9	158.6	161.6	134.9	266.5	9.8	76.2

`ccrypt`, `exif`, and `gcc`, finding the top predictor while instrumenting less than 20% of sites on average.

Table 4.3 shows the mean number of sites instrumented during an iteration. Table 4.4 shows the mean number of iterations required to find the top bug predictor. The columns show the various scoring heuristics. The column titled BwImp corresponds to backwards analysis performed using the *Importance* heuristic. *BFS*'s wide fan-out reveals the top predictor in fewer iterations but instruments many sites. Other forward heuristics instrument roughly the same number of sites per iteration but differ in the number of iterations required. `flex` has a relatively flat CDG due to a large number of *switch* statements in its input scanner. This causes significantly many sites to be instrumented per iteration. The best predictor for `bc` is very close to the point of failure and hence backward analysis completes quickly. The number

of iterations required is in the order of tens for Siemens programs and in the order of hundreds for large applications, which is not large for wide deployments that generate many feedback reports.

Caveats regarding Instrumentation Selectivity

Two factors in the above discussion are worth further consideration. First, the *Importance* score assigned to predicates during adaptive bug isolation will be different from the score computed using non-adaptive bug isolation. However, as more and more data is collected, the scores assigned by both the approaches will get close to each other. Even though we stop instrumentation of a predicate at the end of the iteration in which it was chosen, the fact that we do not use sampling helps. Our technique can gather enough samples for each predicate quicker than sampled, non-adaptive instrumentation.

Second, the number of iterations needed for finding the top predictor varies anywhere from 18 to 650. However, consider the number of feedback reports needed to gather sufficient observations of a predicate. This number will be significantly smaller for non-sampled, adaptive instrumentation as compared to sampled, non-adaptive instrumentation. Although a gross oversimplification, the intuition is that gathering sufficient observations of a predicate while sampling at a rate of $1/100$ would need a hundred times as many reports as non-sampled data collection. Hence, the overall time needed by our technique to produce useful results would roughly be of the same order of magnitude as sampled, non-adaptive instrumentation.

Given that our technique omits monitoring a vast majority of predicates during every run, it is not possible to improve upon, or even match, the number of feedback reports or time needed for fault localization. We can, however, match the failure predictors found by non-adaptive instrumentation. The takeaway from this section of experiments is that, by selecting an appropriate candidate for *WaitForSufficientData*, we can perform bug isolation using a reasonable number of feedback reports, while using extremely lightweight instrumentation (as seen later in section 4.4.4).

4.4.3 Multiple Bugs

If there are multiple bugs in the program, adaptive analysis might exclusively select predicates relevant to the most relevant bug. Or, it might keep selecting predicates relevant to several different, but equally

Table 4.5: Results for `exif`

Bug	Number of Failed Executions	Single Instance		Parallel	
		Iterations	Sites	Iterations	Sites
`libjpeg`	228	32	100	41	120
`exif-print`	180	423	817	26	68

prevalent bugs. Neither is desirable. Infrequent bugs are ignored in the first case, while it takes longer to find good predictors for any of the bugs in the second case. To handle these pathologies, we propose a solution similar in spirit to that of the iterative bug isolation algorithm in section 2.3.2. The failed runs in the feedback reports can be clustered based on the circumstances in which they fail. This requires grouping failed runs by cause. For crashes, one may use crash stacks or just the crashing program counter to label failures. The set of failed runs is divided into disjoint sets, each labeled with the failing program counter or crash stack. Runs with a different label can be ignored while attempting to find good bug predicates for a particular set of labeled failures. Based on this intuition, multiple instances of procedure 1 are run. Each instance considers one set of labeled failures, and all the successful runs. This solution avoids both the pitfalls mentioned above. Deployed programs can be randomly split to collect feedback data for these multiple instances.

To illustrate how programs with multiple bugs are handled, we return to our `exif` benchmark. As mentioned earlier in the chapter, `exif` contains three known bugs. The bug in the module processing Canon images was exhibited only once in our test suite of 10,000 runs and, hence, is ignored. One of the bugs is in the `libjpeg` module. The other bug is in the printing code in `exif`. The bugs cause 228 and 180 runs to fail respectively. Table 4.5 shows the number of iterations and predicates instrumented before which the single-instance and parallel-instance variants of adaptivity find the top bug predictor. A single instance of procedure 1 finds the top bug predictor for the `libjpeg` bug in about 32 iterations, after instrumenting just 100 of 2,631 sites. The best predictor for the `exif-print` bug is not found until iteration 423, after instrumenting 817 sites. Manual inspection shows that many sites instrumented between iterations 32 and 423 relate to the first bug. This affirms that adaptive instrumentation can stall in the presence of multiple bugs. Per our proposal above, each failed run for `exif` is labeled based on

Table 4.6: Relative performance overheads

	Sampling				Adaptive	
Program	1/1	1/100	1/UINT_MAX	Binary	TTest	BFS
`bash`	1.315	1.257	1.137	1.857	1.127	1.155
`bc`	1.172	1.146	1.130	1.271	1.001	1.039
`gcc`	3.686	2.426	1.651	7.261	1.001	1.036
`gzip`	3.583	2.012	1.565	3.425	1.021	1.094
`exif`	1.893	1.975	1.292	6.726	1.003	1.060
Overall	2.076	1.692	1.338	3.305	1.030	1.076

the program counter in which it crashes. We run two separate instances of procedure 1. Each instance analyzes the failed runs from one labeled-set of failures and all the successful runs. With this parallel instantiation, the best bug predictor for the first bug is found in 41 iterations and 120 instrumentation sites. The second bug is caught after 26 iterations and 68 sites. This shows that, while procedure 1 by itself is not designed for multiple bugs, it can be easily modified to handle them.

4.4.4 Performance Impact

Table 4.3 shows that very few sites are instrumented at any time. We might then expect lower overheads. We test this hypothesis by measuring the mean overhead for executing the monitoring plans suggested by the *TTest* and *BFS* heuristics. Table 4.6 shows measured overheads relative to 1 for non-instrumented code. We evaluate performance for `bash`, `bc` and `exif`, whose non-instrumented programs run for approximately 0.25, 4, and 0.01 seconds, respectively. The test suites for other programs, being functionality tests rather than performance tests, execute for extremely short periods (order of a few milliseconds) and their performance cannot be reliably measured. For `gcc` and `gzip`, we evaluate performance using inputs in the SPEC benchmark suite. Non-instrumented programs take between 0.5 and 50 seconds on these inputs. We compare our technique against complete binary instrumentation and the sampling scheme of Liblit et al. [Lib+05]. We experiment with sampling rates of 1/UINT_MAX, 1/100, and 1/1. A sampling rate of 1/UINT_MAX is the best case for sampling based instrumentation. The execution continues along the fast-path almost all the time. A rate of 1/100 has been suggested by prior work [Lib+03] for public deployments.

Adaptive instrumentation is at least an order of magnitude faster than complete binary instrumentation and significantly faster than all sampling variants. *BFS*, due to its wide fan-out, has a higher overhead than *TTest*. The overhead is minuscule for `bc`, `gcc` and `gzip`, where long running times amortize instrumentation costs. The mean overheads for adaptive instrumentation is 1.03× for adaptive instrumentation, as opposed to 1.69× for sampling at a rate of $1/100$, and 3.31× for complete binary instrumentation. Overheads for `bash` and `exif` are easily affected by measurement noise as even a 0.01 second offset could change the overhead by at least 4%. If these short-lived programs are excluded, the average overhead for the remaining programs is 3.2× for complete binary instrumentation, 1.78× for sampling at a rate of $1/100$ and just 1.008× for adaptive instrumentation. An overhead of less than 1% is effectively imperceptible to an end-user. Sampling, however, imposes an overhead of 1.34× even at its best-case sampling rate of $1/\mathrm{UINT_MAX}$. Selective instrumentation, if applied by a static instrumentor, can achieve order-of-magnitude improvements over sampling. But, as mentioned earlier, enacting new plans would require distributing executable patches: an impractically resource-intensive proposition.

4.4.5 Comparison with Holmes

The Holmes project by Chilimbi et al. [Chi+09] introduces two orthogonal concepts: path-based instrumentation and adaptive predicate selection. It introduces a new predicate scheme that counts the number of times each path is taken in an acyclic region. The adaptation step of Holmes proceeds as follows: based on partial feedback data, weak predictors are selected, and functions close to them in the program-dependence graph are chosen. Predicates in these functions are instrumented during the next iteration. In this chapter, we develop and evaluate a heuristic search at a much finer granularity. Holmes also strengthens weak predictors by selecting path predicates in functions containing weak branch predicates. This strengthening is orthogonal to the heuristic searches proposed here or in Holmes.

In this section, we compare our technique with the adaptive component of Holmes. We do not consider path predicates for two reasons. First, our binary and source instrumentors do not yet support the preferential path profiling technique [Vas+07] used to efficiently implement path predicates. Second, Holmes's use of path predicates is orthogonal to adaptivity, and could be added to our system in the same

manner.

Holmes defines *Vicinity* used in procedure 1 at the granularity of entire functions. Holmes finds *weak predictors* instrumented in earlier iterations, defined as predicates with *Importance* scores between 0.5 and 0.75. It selects functions close to these predictors in the PDG, and instruments all predicates in these neighboring functions in the next iteration.

Weak predictors can be quite sparse. Because Holmes explores only near weak predictors, this creates a risk that it can get stuck with no new sites available to explore. In our experiments, this was the case in 46 of the 130 Siemens experiments and 61 of the 97 larger experiments. In 111 of the remaining 120 experiments in which Holmes finds the top predictor, the sparsity of weak predictors is side-stepped because the top predictor is so close to the point of failure that it is instrumented in the very first iteration. While the definition of weak predictors seems to be the impediment here, if we remove that restriction and define *Vicinity* to explore near all predictors, then Holmes reduces to doing a breadth-first search on the call graph, which is a coarser version of the *BFS* heuristic evaluated earlier. Our approach cannot get stuck, as noted in section 4.3.4.

In general, exploring at the granularity of functions is coarser than *BFS*. Hence, more sites will be instrumented per iteration, imposing more overhead, but requiring fewer iterations. Our approach is more flexible, allowing trade-offs between overhead and the number of iterations. We do not present a direct performance comparison because Holmes performs only trivial (single-iteration) explorations of all programs in table 4.6.

4.5 Summary

Post-deployment bug hunting is a search for a needle in a haystack. Monitoring strategies that cannot respond to feedback incur large overheads and waste considerable computational resources. In this chapter, we have developed an adaptive post-deployment monitoring system using statistical analysis, static program structure, and binary instrumentation. Of several search heuristics considered, one (*TTest*) consistently performs well in the forward direction while another (*Importance*) shows promise when working backward from known points of failure. We find that this technique achieves the same results as

an existing statistical method while monitoring, on average, just 40% of potential instrumentation sites in the programs we considered. Performance measurements show that our technique imposes an average performance overhead of less than 1% for a class of large applications as opposed to 69% for realistic sampling-based instrumentation. Monitoring overheads are so small as to be nearly immeasurable, making our adaptive approach practical for wide deployment.

Chapter 5

Identifying Failure-inducing Changes

In this chapter, we develop an application that uses the data collected by CBI to help debugging in the context of evolutionary software development. In this context, multiple changes have been made to a program, and some of them could be failure-inducing. Associating failing locations of the program with source-level changes could be useful to the programmer while debugging. Bug predictors produced by statistical analysis can also be associated with source-level changes. The source-level change identifies a potential root-cause of failure and the bug predictor captures the effect of the root-cause that is symptomatic of failure. This association between source-level changes and bug predictors is potentially more useful than each piece of information by itself. Identifying failure-inducing changes is an useful aid while debugging.

To associate failing locations and bug predictors with source-level changes, we can use techniques for change impact analysis [Arn96]. Change impact analysis studies the problem of identifying the components of a program that are potentially impacted by a change to a program. A change is considered to impact a component if for some input to the program, making the change to the program changes the behavior [Hor+88] of the component. Typically, software developers use change impact analysis to understand the impact of modifications, find related parts that also need to be changed, or select test cases that should be re-run to test some change. A wide variety of change impact analyses have been developed to compute impact sets at various granularities: statements [Bin97; Hut+94], functions [Api+05; Kun+94; Ren+04], or files [Yin+04]. Some analyses are entirely static [Bin97; Yin+04], while others use dynamic traces

[Api+05; Hut+94; Kun+94; Ren+04].

Traditionally, the output of a change impact analysis is a Boolean decision on whether a particular program point is impacted. The impact sets found by these techniques could be used to associate bug predictors or failing locations with source-level changes. If the bug predictor or failing location of interest is in the set of impacted locations corresponding to exactly one source-level change, then we can make an association between the location of interest and the source-level change. We performed this association using forward slicing as our impact analysis technique: a program point p is considered to be impacted by a change made at a program point c if and only if p is in the forward slice from c. The exact details of this experiment are presented in section 5.3.3. We found that almost all of the bug predictors and failure locations were in the forward slice from multiple changes. Hence, this classification was not very useful for associating bug predictors and failure locations with source-level changes.

Instead of a Boolean decision on whether a program point is impacted, we propose to quantify the impact of the change on each program point as a more nuanced, probabilistic value. We approximate the likelihood of a change impacting a program location p using the probability that p executes after the change. The intuition behind this approximation is that a change c can impact a location p only if p executes after c. So we hypothesize that the probability that p executes after c is a reasonable measure of the likelihood of c impacting p. We then associate a bug predictor or a failing location with the change that has the highest probability of impacting it. This is only a heuristic for associating bug predictors and failing locations with the "responsible" source-level changes. But, as discussed in section 5.3, we find that this works well in practice.

For a given changed location c, we define a *conditional coverage profile* that contains the probability that any location in the program executes after c. In this chapter, we develop a technique to estimate the conditional coverage profile using branch and call-graph profiles. In profile-guided optimization, a *simple coverage profile* may refer either to the frequency of execution of statements (and paths) or to the edge profiles of function-call and conditional statements. Edge profiles can be used to compute the probability that any program point executes during a typical execution of the program. It is usually approximated by running the program on a test suite. Here, we define a *conditional coverage profile* of a program and a

changed location. We use "conditional" in the sense of conditional probability: the likelihood of some event, conditioned on a related event. Specifically, a conditional coverage profile for a change c represents the probability that any program point executes after the first execution of c. Such a profile can be used to model the propagation of the impact of a code change.

The value assigned to each program point is the probability that it executes after the first execution of the change c. This quantity can be exactly computed for a given program run using direct instrumentation. We can instrument the program to start collecting branch and call-graph profiles after the first execution of the c. This is simple enough, but scales badly for multiple changes. For that level of flexibility, we would need multiple instrumented copies of the program, one for each change c, since we only want to start profiling after c executes. In CBI's regime of post-deployment monitoring, this would translate to a long delay between picking a change c, distributing new binaries, collecting sufficient feedback data, and finally computing the conditional coverage profile. Obviously this is impractical, especially in the extreme case where every program point can be changed. Instead, we propose to use branch and call-graph coverage profiles to approximate the value of a conditional coverage profile. As with all dynamic analyses, poor test suites can yield poor conclusions. However, if the test suite is good, our dynamically-informed approach may give richer guidance than a purely static analysis.

The rest of this chapter is organized as follows. Section 5.1 explains our technique to estimate conditional coverage profiles using branch and call-graph profiles. Section 5.2 describes our experimental setup and presents the time taken for data collection and computation of conditional coverage profiles. Section 5.3 presents and evaluates our technique to associate bug predictors and failing locations with source-level changes.

5.1 Conditional Coverage Estimation

The goal here is to assign a numeric value to each node in a program that captures the likelihood that a source node impacts the line. We formalize this notion in this section.

Definition 6. *Let s be a changed node in the inter-procedural control-flow graph (ICFG). The conditional coverage profile associates each node n n in the ICFG with the value ψ_n, which is the probability that n is*

executed at least once after the first time the control flow reaches s. "After" here is non-strict: s is always considered to execute after itself, even if reached just once.

This is a well-defined quantity for a given set of runs, which can be calculated by instrumenting the program. The program can be instrumented to start collecting a coverage profile after the first execution of the source node s. As with any coverage information, depending on the quality of the test suite, inferences based on the conditional coverage profile can be applied to general program behavior. However, this approach increases the instrumentation overhead linearly with respect to the number of points of interest for which the profile is computed. Instead, we propose to estimate this value from the whole-program coverage profile. Collecting the whole-program coverage profile is a one-time task that can be accomplished efficiently using CBI's sampling-based instrumentor. Of course, this estimate can be imprecise for several reasons. We discuss the sources of imprecision in section 5.1.5

5.1.1 Basic Definitions

Let e_n denote the event that node n executes after the first execution of the source node. Also, for an edge from node m to node n in the control-flow graph, let e_{m-n} denote the event that the edge is traversed after the first execution of the source node. The union of edge events a and b occurs when either event a or b occurs after the first execution of the source node. The intersection of events a and b occurs when both a and b occur after the first execution of the source node. We use the standard notation $P(\cdot)$ to denote the probability of an event. Then, $\psi_n = P(e_n)$. Then, the probability of the event e_{m-n} can be expressed as follows:

$$P(e_{m-n}) = \begin{cases} b_{m-n} \times P(e_m) & \text{if } m \text{ is a branch node} \\ P(e_m) & \text{otherwise} \end{cases} \tag{5.1}$$

where b_{m-n} is the fraction of times that control flows to n after control reaches m. So, if $succ(m)$ denotes the successors of m in the control-flow graph, then $\sum_{n \in succ(m)} b_{m-n} = 1$. The case in Equation (5.1) where the predecessor m is a branch node conservatively assumes that m executes only once after the first execution of the source node. This conservative assumption can lead to under-approximations when m executes more than once. For example, consider executions where m executes exactly twice after the first execution

of s during every execution. The probability that the edge $m-n$ is not taken during one execution of the branch statement at m is $1-b_{m-n}$. The probability that the edge $m-n$ is not taken during both of the executions of m is equal to $(1-b_{m-n})^2$. Then, the probability that the edge $m-n$ executes at least once is $1-(1-b_{m-n})^2 = b_{m-n} \times (2-b_{m-n})$. This is greater than the value, b_{m-n}, that would be assigned by equation (5.1). Such under-approximations can be avoided if we keep track of fine-grained probabilities such as:

$P(m$ executes exactly k times after the first execution of $s)$ for k = 0, 1, 2,…

Such fine-grained estimations will complicate the comparison operations needed later in this chapter. Instead, we use the under-approximation in equation (5.1).

Let $pred(n)$ denote the predecessors of node n in the control-flow graph. Then, the probability that n executes after the first execution of the source s is defined recursively as follows:

$$\psi_s = 1 \tag{5.2}$$

$$\psi_n = P(e_n) = P\left(\bigcup_{m \in pred(n)} e_{m-n}\right) \tag{5.3}$$

Equation (5.2) is the base case: s always executes (non-strictly) after itself. For any other node n, execution reaches n after the first execution of s when any of the *in-edges* of n in the control-flow graph is executed after s. Thus, the probability of the event e_n is the probability of the union of events e_{m-n} for all CFG predecessors m of n.

If n has only two predecessors v and w, then ψ_n simplifies to

$$\psi_n = P(e_{v-n} \cup e_{w-n}) \tag{5.4}$$

$$= P(e_{v-n}) + P(e_{w-n}) - P(e_{v-n} \cap e_{w-n}) \tag{5.5}$$

5.1.2 Estimating Co-occurrence of Nodes

The term $P(e_{v-n} \cap e_{w-n})$ in equation (5.5) can be measured by direct instrumentation. In this section, we explain how we can estimate this quantity instead, thereby avoiding the cost of instrumentation. If we can show that the executions of v and w are mutually exclusive, then $P(e_{v-n} \cap e_{w-n}) = 0$ and equation (5.5) simplifies to

$$\psi_n = P(e_{v-n}) + P(e_{w-n})$$

For the general case where execution of the edges in $pred(n)$ are all mutually,

$$\psi_n = \sum_{m \in pred(n)} P(e_{m-n})$$

However, if the execution of the edges is not mutually exclusive, we can only estimate a lower bound on $P(e_{v-n} \cup e_{w-n})$. Suppose, $P(e_{v-n}) < P(e_{w-n})$, that is, the edge $w-n$ has a higher probability of execution after the source than $v-n$. The conservative assumption while estimating the lower bound for $P(e_{v-n} \cup e_{w-n})$ would be that the event e_{v-n} occurs only when e_{w-n} also occurs. In this case, the probability of the intersection of the two events is the probability of the rarer event, i.e. $P(e_{v-n})$. From equation (5.5), the probability of the union of the two events reduces to $P(e_{w-n})$. More generally,

$$P(e_{v-n} \cap e_{w-n}) \leq \mathbf{min}(P(e_{v-n}), P(e_{w-n}))$$
$$\psi_n = P(e_{v-n} \cup e_{w-n}) \geq \mathbf{max}(P(e_{v-n}), P(e_{w-n}))$$

We approximate ψ_n using the lower bound above:

$$\psi_n \approx \mathbf{max}(P(e_{v-n}), P(e_{w-n})) \tag{5.6}$$

A predecessor m of a node n falls into one of two categories depending on whether the edge $m-n$ is a back edge. Let $f_pred(n)$ be the set of predecessors $m \in pred(n)$ such that the edge $m-n$ is a forward edge. Let $b_pred(n)$ be the set of predecessors $m \in pred(n)$ such that $m-n$ is a back edge, i.e. $b_pred(n) = pred(n) \setminus f_pred(n)$. For the forward edges reaching a node n, only one of them can be

executed during one execution of the acyclic region. As explained in the discussion about equation (5.1), we are unable to distinguish among cases where nodes execute more than once. So, we conservatively assume that they execute only once. In this case, the forward edges execute mutually exclusively of each other. Likewise, the back edges reaching n execute mutually exclusively of each other. However, a forward edge reaching n is not mutually exclusive of a back edge reaching n. So, we can merge the forward and back edges as a unified edge, and compute ψ_n^f and ψ_n^b, respectively the probabilities that the execution reaches n via forward and back edges respectively. Using the same reasoning while deriving equation (5.6), the best estimate of ψ_n is the maximum of ψ_n^f and ψ_n^b.

$$\psi_n^f = P(\bigcup_{t \in f_pred(n)} e_{t-n}) = \sum_{t \in f_pred(n)} P(e_{t-n}) \tag{5.7}$$

$$\psi_n^b = P(\bigcup_{t \in b_pred(n)} e_{t-n}) = \sum_{t \in b_pred(n)} P(e_{t-n}) \tag{5.8}$$

$$\psi_n \approx \mathbf{max}(\psi_n^f, \psi_n^b) \tag{5.9}$$

In a reducible control graph, back-edges of a loop execute in a subset of the executions that reach the loop header. In a *general* coverage profile that is not conditioned on the prior execution of another node, the probability of the execution reaching n via the forward edges is always greater than the probability of the execution reaching n via the back edges: $\psi_n^f > \psi_n^b$. This will not hold true in conditional coverage profiles. If the source node of the conditional coverage profile is in the loop body, or in a function called from the loop body, the probability of the execution reaching n via back edges may be greater than the probability of the execution reaching n via forward edges.

For irreducible control-flow graphs, it is not possible to divide the control-flow edges into a disjoint set of forward and back edges. In the case of irreducible CFGs, we have to assume that none of the predecessors are mutually exclusive, and use the highest probability along an incoming edge as the best estimate. That is,

$$\psi_n \approx \mathbf{max}_{t \in pred(n)} (P(e_{t-n}))$$

This situation does not arise in any of our experiments.

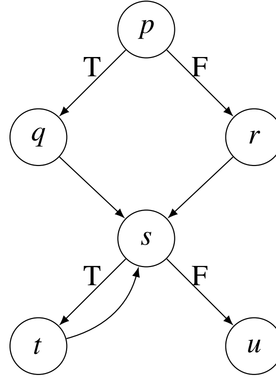


Figure 5.1: Example control-flow graph fragment

To recap the concepts introduced so far, consider the CFG fragment in fig. 5.1. There are two branch nodes, p and s. The probabilities of their children are obtained using the branch profiles as follows:

$$\psi_q = P(e_{p-q}) = b_{p-q} \times \psi_p \qquad \psi_t = P(e_{s-t}) = b_{s-t} \times \psi_s$$

$$\psi_r = P(e_{p-r}) = b_{p-r} \times \psi_p \qquad \psi_u = P(e_{s-u}) = b_{s-u} \times \psi_s$$

The node s has three predecessors: q, r, and t. The edges $q-s$ and $r-s$ are forward edges, while $t-s$ is a back edge. Using the derivation for the multiple-predecessor scenario above,

$$\begin{aligned}
\psi_s^f &= P(e_{q-s} \cup e_{r-s}) = P(e_{q-s}) + P(e_{r-s}) \\
&= P(e_q) + P(e_s) = \psi_q + \psi_s \\
\psi_s^b &= P(e_{t-s}) = \psi_t \\
\psi_(s) &\approx \mathbf{max}(\psi_s^f, \psi_s^b) \\
&= \mathbf{max}(\psi_q + \psi_r, \psi_t)
\end{aligned}$$

As mentioned earlier, which of ψ_s^f, ψ_s^b is greater depends on the source node used for the conditional coverage profile. If t, or a node in a function called from t, is the source, the probability that the edge $t-s$ is taken after the first execution of the source will be higher than the combined probability that edges $q-s$ and $r-s$ are taken.

In this section, we define ψ_n for a node n in terms of the value of its predecessors, except for the base case in equation (5.2). These definitions are independent of the choice of the source node of the conditional coverage profile. They are statically defined based on the structure of the ICFG. The conditional coverage profile for a chosen source node s is computed by adding equation (5.2) and simplifying the equations. For completeness, ψ_e is set to 0, where e is the program entry node. This is done to derive values for nodes that are not in the control-flow closure from the source node, and hence have a value of 0 in the conditional-coverage profile.

5.1.3 Handling Function Calls

Two issues arise while extending our estimation of conditional coverage profiles across procedures. First, execution probabilities have to be propagated from the function with the source node transitively to all possible callers of that function. The rest of this section addresses this first issue. The second issue, recursion, is addressed in section 5.1.4.

To understand the first issue, consider the procedure f that contains the source node s. After the first execution of the source node s, execution continues along other statements in f, eventually returning to one of f's callers. We use call-graph profiles to compute the fraction of executions that return to each caller. Our representation of the inter-procedural control-flow graph has a *return* node associated with each function call node. Let $caller(f)$ be the set of nodes that call f. For a node n in $caller(f)$, let n^r be the associated return node. For a node $n \in caller(f)$, let c_{n-f} be the fraction of calls to f that arise from the call statement at n. Then the fraction of executions that execute s before returning from f to n is

$$P(n^r) = P(s) \times c_{n-f} \qquad (5.10)$$

The above equation initializes $P(n^r)$ rather than $P(n)$ to capture the control flowing to the *return* node rather than the function call node. Each node n^r is now transitively considered a source node, and the above definition is re-applied in that context. The values c_{n-f} are computed by extending the CBI instrumentor to collect call-graph profiles. For a node $n \in caller(f)$, let e_{n-f} be the number of times the call edge from n to f is observed across all executions in the test suite. Then, c_{n-f} is obtained by normalizing e_{n-f} over

all callers of f.

$$c_{n-f} = \frac{e_{n-f}}{\sum\limits_{k \in caller(f)} e_{k-f}} \tag{5.11}$$

The instrumentation to profile call edges is integrated with the existing sampling framework. Sampling helps us reduce the overhead of data collection. When sampling is enabled, the execution counts of call edges are a fair random sample of actual calls invoked. Since we normalize the counts collected using sampling, the computed call-graph profiles get closer in accuracy to the true profiles as data accumulates from more runs. We also instrument indirect function calls while collecting call-graph profiles. We use a points-to analysis to identify all possible callees at each indirect call.

5.1.4 Recursion and Loops

So far in this section, we have been defining the conditional coverage profile ψ_n for a node n in terms of the corresponding values of its predecessor nodes. These definitions are circular in the presence of recursion and loops. At the sources of such circularity (recursive calls and loop headers), symbolic values are introduced. We have built a symbolic expression utility that can handle linear symbolic expressions, implements addition of two expressions, addition and multiplication by a scalar, and comparison. We use a linear-algebra solver to solve the set of linear constraints. Comparison is required for the **max** operation used in equation (5.6). Consider two symbolic expression A and B. Since the value of each symbolic term is non-negative, $A \geq B$ if each term in A has an equal or higher coefficient than the corresponding term in B. For instance, $A = 0.5x + 0.9y$ is greater than $B = 0.5x + 0.2y$. On the other hand, $0.5x + 1.0$ is not comparable with $0.9y + 0.8$. When two expressions are incomparable, we use two techniques. First, we aggressively try to evaluate all expressions any time a comparison is performed. This might simplify one or both the expressions, making comparison possible. Secondly, when unable to evaluate both arguments of the comparison, we return the expression with the highest coefficient among all its terms. This is just a heuristic, and is another source of imprecision in our analysis.

To illustrate the concepts listed here, consider the control-flow graph in fig. 5.2. This corresponds to a simple program that computes the integral part of the square root of an integer. The most interesting section of the CFG is the `while` loop in function `sqrt` that increments an iterator variable `r` until `r × r` exceeds

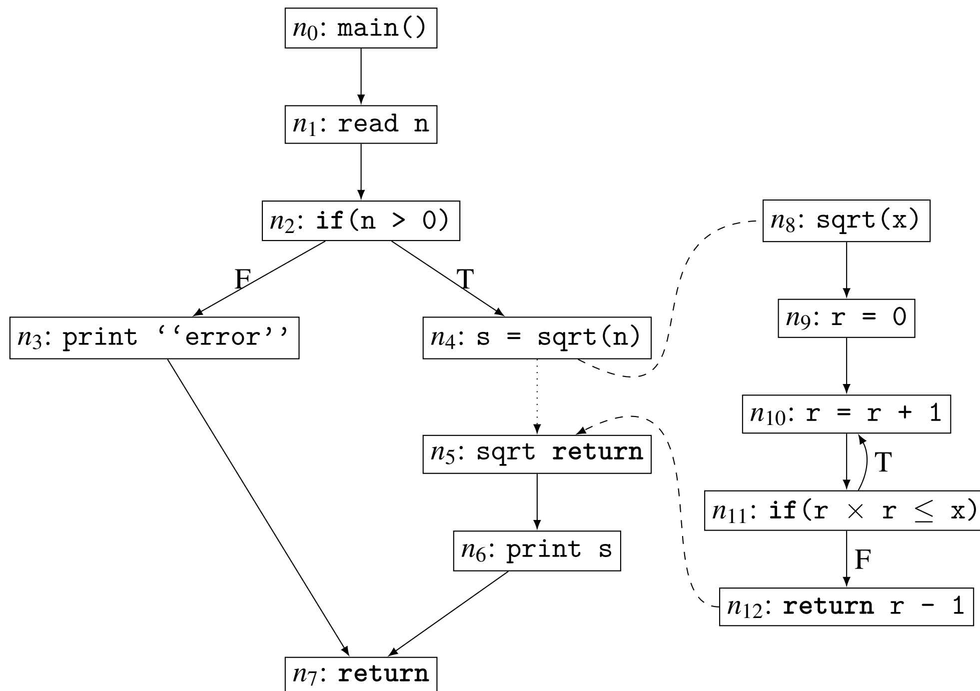


Figure 5.2: Control-flow graph for a simple square root computation

the input parameter. The node n_5 represents the point in `main` to which execution returns after completing the call to `sqrt` that started at node n_4 The equations governing all conditional coverage profiles for this code are listed in fig. 5.3. Figure 5.4 shows the conditional coverage profile when node n_1 is the source. This is achieved by setting ψ_{n_1} to 1, ψ_{n_0} to 0 and simplifying the equations in fig. 5.3. One non-trivial simplification in fig. 5.4 is the application of the **max** operator for $\psi_{n_{10}}$. Since the two expressions are not directly comparable, we fall-back on the heuristic of picking the expression with the highest coefficient among its terms. Here, the heuristic selects $b_{n_2-n_4}$ since the other option would lead to $\psi_{n_{10}}$ being assigned $b_{n_{11}-n_{10}} \times \psi_{n_{10}}$ and consequently, 0. Figure 5.5 shows the conditional coverage profile when n_{11} is the source. This is obtained by adding the equations $\psi_{n_{11}} = 1$ and $\psi_{n_0} = 0$. Per the interprocedural step in section 5.1.3, we also add $\psi_{n_5} = 1$ since n_5 is the *return* node corresponding to the function call at n_4.

$$\psi_{n_2} = \psi_{n_1} = \psi_{n_0}$$
$$\psi_{n_3} = b_{n_2-n_3} \times \psi_{n_2}$$
$$\psi_{n_4} = b_{n_2-n_4} \times \psi_{n_2}$$
$$\psi_{n_6} = \psi_{n_5} = \psi_{n_4}$$
$$\psi_{n_7} = \psi_{n_3} + \psi_{n_6}$$
$$\psi_{n_9} = \psi_{n_8} = \psi_{n_4}$$
$$\psi_{n_{10}} \approx \mathbf{max}(\psi_{n_9}, b_{n_{11}-n_{10}} \times \psi_{n_{11}})$$
$$\psi_{n_{11}} = \psi_{n_{10}}$$
$$\psi_{n_{12}} = b_{n_{11}-n_{12}} \times \psi_{n_{11}}$$

Figure 5.3: General equations for conditional coverage profile for the CFG in fig. 5.2

$$\psi_{n_2} = \psi_{n_1} = 1$$
$$\psi_{n_3} = b_{n_2-n_3}$$
$$\begin{aligned}\psi_{n_4} &= b_{n_2-n_4} \times \psi_{n_2} \\ &= b_{n_2-n_4}\end{aligned}$$
$$\psi_{n_6} = \psi_{n_5} = b_{n_2-n_4}$$
$$\psi_{n_7} = \psi_{n_3} + \psi_{n_6} = 1$$
$$\psi_{n_9} = \psi_{n_8} = b_{n_2-n_4}$$
$$\begin{aligned}\psi_{n_{10}} &\approx \mathbf{max}(b_{n_2-n_4}, b_{n_{11}-n_{10}} \times \psi_{n_{10}}) \\ &\approx b_{n_2-n_4}\end{aligned}$$
$$\begin{aligned}\psi_{n_{11}} &= \psi_{n_{10}} \\ &\approx b_{n_2-n_4}\end{aligned}$$
$$\begin{aligned}\psi_{n_{12}} &= b_{n_{11}-n_{12}} \times \psi_{n_{11}} \\ &\approx b_{n_{11}-n_{12}} \times b_{n_2-n_4}\end{aligned}$$

Figure 5.4: Conditional coverage profile from node n_1 for the CFG in fig. 5.2

$$\psi_{n_2} = \psi_{n_1} = \psi_{n_0} = 0$$
$$\psi_{n_3} = b_{n_2-n_3} \times 0 = 0$$
$$\psi_{n_4} = b_{n_2-n_4} \times 0 = 0$$
$$\psi_{n_6} = \psi_{n_5} = 1$$
$$\psi_{n_7} = \psi_{n_3} + \psi_{n_6} = 1$$
$$\psi_{n_9} = \psi_{n_8} = 0$$
$$\psi_{n_{10}} \approx \mathbf{max}(0, b_{n_{11}-n_{10}} \times 1) = b_{n_{11}-n_{10}}$$
$$\psi_{n_{11}} = 1$$
$$\begin{aligned}\psi_{n_{12}} &= b_{n_{11}-n_{12}} \times \psi_{n_{11}} \\ &= b_{n_{11}-n_{12}}\end{aligned}$$

Figure 5.5: Conditional coverage profile from node n_{11} for the CFG in fig. 5.2

5.1.5 Sources of Imprecision

The sources of imprecision in our estimation of conditional coverage profiles can be classified broadly into two kinds. The first kind arises due to the restrictions imposed by our problem definition. We assume that control-flow profiles are computed with a composite set of changes enabled. If we instead have the ability to enable each change individually, we can calculate the conditional coverage profile more precisely. Moreover, the behavior of branches before and after the first execution of the source node can be different. We also lose context sensitivity and path sensitivity by using branch profiles instead of path profiles. The value for $\psi_{n_{12}}$ in fig. 5.5 shows another reason why using branch profiles are imprecise. While node n_{12} always executes after the first execution of n_{11} in a terminating execution, we only assign a value of $b_{n_{11}-n_{12}}$ to $\psi_{n_{12}}$. This can be solved by using a different definition of branch profiles. If we define $b'_{n_{11}-n_{12}}$ as the fraction of runs in which the edge $n_{11} - n_{12}$ is taken, it will have a value 1 since this edge is always crossed in terminating executions. However, this definition does not work well when run-time data is sampled rather than being collected exactly and completely. Since the edge is taken only once during each execution of the `while` loop, it is unlikely to be observed during sparse random sampling. Since we need sampling to ensure low overheads, we stick to our original definition of branch profiles.

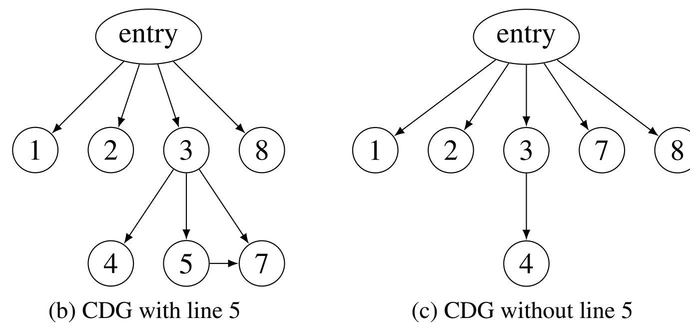
The second broad class of imprecisions arises from our goal to reduce the performance overhead needed to collect these profiles. Any or all of this second class of imprecisions can be eliminated in exchange for higher overheads. We use sampling to collect a fair random sample of the branch profiles. We can compute $P(e_{v-n} \cap e_{w-n})$ used in equation (5.5) directly by instrumentation, but the instrumentation overhead imposed will be high. This may be preferred during in-house testing, when accuracy trumps speed. Our current implementation allows only elimination of sampling imprecision.

5.1.6 Change Impact Analysis

When a change is made to a program, it may also negatively impact whether a node executes after a change. Before a proposed change is made to a node s, a node can have a high probability in the coverage profile from s. The same node may have a low probability in the coverage profile after the proposed change is made. This can happen due to changes in the branch profiles of conditional statements that are transitively

```
1 a = 2;
2 b = 2;
3 if (a == 2) {
4    b = 6;
5    goto ret; /* deleted code */
6 }
7 b = 4;
8 ret: print(b);
```

(a) Code snippet



(b) CDG with line 5 (c) CDG without line 5

Figure 5.6: Example illustrating change impact analysis

dependent on the changed node. We also consider such a reduction in the probability of execution as an impact. Therefore, when we use conditional coverage profiles to capture the effect of a change on a node in the program, we set the probability of impact as the maximum of the conditional coverage from the changed nodes in the old and new versions.

Similarly, when using slicing for change impact analysis, both the old and the new versions must be considered to handle code deletions. To illustrate this case, consider the code snippet in fig. 5.6a and corresponding control-dependence graph in fig. 5.6b. Lines 1, 2, 3 and 8 are always executed, and hence dependent on the entry node. Lines 4 and 5 execute only when the branch condition at line 3 is true. Line 7 executes only when the branch condition at line 3 is false. Thus, lines 4, 5 and 7 are control-dependent on line 3. Moreover, there is a control-dependence edge from line 5 to line 7 because the execution of line 5 prevents execution of line 7. Consider the atomic change of removing line 5 from the snippet. Figure 5.6c shows the control-dependence graph of the new version, after the change is applied. The dependency from line 5 to line 7 does not exist in the new control-dependence graph because the node corresponding to

Table 5.1: Bug benchmarks used in experiments

Program	Size		Change Count
	Lines of Code	Test Cases	
`flex` v1	14,725	567	6
`flex` v2	15,265	567	4
`flex` v3	15,303	567	3
`flex` v4	16,967	567	3
`flex` v5	15,367	567	2
`grep` v1	14,775	809	3
`grep` v2	15,489	809	3
`grep` v3	15,676	809	3
`grep` v4	15,721	809	3
`gzip` v1	7,348	217	3
`gzip` v2	8,014	217	2
`gzip` v5	8,960	217	3
`sed` v2	13,177	365	4
`sed` v3	11,229	365	4
`sed` v5	18,063	365	4
`sed` v7	19,774	365	3

line 5 does not exist in the new CDG. Thus, line 7 is not in the static slice from the set of changed nodes because the set of changed nodes is empty in this case. However, consider a scalar-pairs predicate defined on line 7 that compares the old value of variable `b` with the new value of `b`. It will change its behavior in the set of runs where the true branch is taken on line 3. In order to compute these missing dependences, we compute the slices separately in the two versions. A predicate is considered impacted by the change if it is in the slice from either the old version or the new version.

5.2 Experimental Setup

Before describing our main application of conditional coverage profiles, which is associating bug predictors and failing locations with source-level changes, we briefly describe our experimental setup. We use the following bug benchmarks obtained from the Software-artifact Infrastructure Repository [Rot+06], all written in C: `flex`, `grep`, `gzip`, and `sed`. There are multiple versions of these benchmarks and each

Table 5.2: Overhead of data collection and time for analysis

		Overhead		Analysis Time (in sec)	
Application	Time per Test Case (in ms)	Base	Extended	Building Graphs	Conditional Coverage
`flex`	2.95	1.42	1.43	34	4
`grep`	2.75	1.44	1.43	565	7
`gzip`	4.98	1.41	1.41	154	6
`sed`	2.06	1.61	1.62	478	5
Overall	2.78	1.46	1.47	100	5

version has multiple changes. Each change is a sequence of straight-line code that is deleted, added, or modified. The changes can be enabled individually and independently of each other. The benchmarks also have associated test suites. The changes cause some of the test cases in the suite to fail. Like experiments in earlier chapters, we consider an abnormal exit or a difference in the output printed by the program as a failure. Since statistical debugging algorithms usually require a sizable sample of failed runs to be effective, we retain only those changes that have more than five failing test inputs for experiments in section 5.3. Table 5.1 lists the various versions of the test subjects, their size (LOC), the size of their test suite, and the number of changes that cause more than five test cases to fail. In all, 53 distinct variants are used for fault-localization experiments.

For each change, we identify its corresponding nodes in the ICFG as follows. We use the `diff` utility program to compare the source code for the faulty and reference variants of each program. We collect the first line in each contiguous sequence of lines in the textual difference. The program nodes that are defined in these locations are considered as the changed nodes for computing conditional coverage profiles.

5.2.1 Performance

In this section, we measure the performance impact of our extensions to the CBI instrumentor. The feedback reports for experiments in this chapter are collected by enabling branches, returns, and scalar-pairs instrumentation schemes. Prior work has shown how these can be implemented with very low overhead using sparse sampling [Lib+03] and adaptive instrumentation (chapter 4). We extend the CBI

instrumentor to collect detailed runtime profiles. In addition to branch profiles collected as a part of the branch instrumentation scheme, profiles of `switch` statements are collected. For use in the interprocedural scenario in section 5.1.3, profiles of function-call edges, including indirect calls, are collected. This extended instrumentation is integrated with the sampling framework of the instrumentor: sparse random call-graph profiles and `switch`-statement profiles are collected when sampling is enabled. Table 5.2 shows the execution overheads for our test subjects. The average execution time across all applications is 2.78ms per test case. The extended instrumentor, that collects call-graph and `switch` profiles, imposes between 1.41× and 1.62× overheads. The extra overhead imposed by our extended instrumentor, as compared to the base instrumentor, is negligible. The base instrumentor imposes an average overhead of 1.46×. The high value is partially due to the low running time of each test case.

Table 5.2 also shows the time taken to compute conditional coverage profiles. It takes between 34 seconds and 10 minutes to build the call graph and control-flow graphs, and resolve indirect function calls using CodeSurfer [Gra06]. The bulk of this time is spent performing pointer analysis to resolve indirect calls. This task is not in the critical path because it uses only the program source. The computation of conditional coverage profile takes between 4 and 7 seconds per changed location. This time includes the construction of the basic equations from the static structure of the ICFG, and simplifying the equations.

5.3 Isolating Failure-inducing Changes

We now describe and evaluate our main application of conditional coverage profiles: augmenting the output of CBI by associating each bug predictor and failing location with a source-level change. Consider two consecutive versions of an application, and suppose $c_1, c_2, \ldots, c_m$ are the changes made to the old version of the program to obtain the new version. For each node associated with a bug predictor and each node associated with a failing location, we wish to find the atomic change that is the most likely to impact this node. To do this, we choose the change such that the node has a higher value in the conditional coverage profile from this change than the profiles from all other changes. Although m may be large, the technique of section 5.1 allows us to compute m approximate coverage profiles from one set of runs with just one instrumented executable.

We extend the notation used to represent the conditional coverage profiles to include the location of the changed node.

Definition 7. *Given a set of atomic changes* $\{c_1, c_2, \ldots, c_m\}$ *in the program, let* ψ_n^i *denote the probability that the node n executes after the first execution of any of the source nodes corresponding to the atomic change* c_i.

For a node n, we find the atomic change after which n is most likely to execute. We identify $change(n)$, the change most likely to impact n as follows:

$$change(n) = c_i \text{ such that } \psi_n^i > \psi_n^j \ \forall j \neq i \tag{5.12}$$

In some cases, ψ_n^i might have the same maximal value for more than one value of i. This can arise for two reasons. First, the nodes corresponding to multiple atomic changes may be close to each other in the PDG. Thus, the values in the conditional coverage profile may be the same for these atomic changes. Second, nodes that are close to the `main` function of the program have a high probability of execution in any run and thus the values in the conditional coverage profile may be the same for all changes. For nodes where such a tie occurs, we do not associate any change with them. Therefore, our evaluation in this section considers both the precision and recall of our technique.

5.3.1 Associating Changes with Predictors

Suppose p is a bug predictor found by statistical debugging. Let $node(p)$ be the node corresponding to predictor p. We select this node arbitrarily from among the program nodes defined in the same source line as the predictor. A preprocessing step in our instrumentor rewrites the program so that each line performs only one significant operation. This ensures that all the nodes in a source line have the same values in the conditional coverage profile. We classify $change(node(p))$ as the most likely cause of the bug predicted by p.

To evaluate this classification, we use the retained changes for each test subject listed in table 5.1. For ground truth, we enable each change c_i separately, and associate the bug predictors identified by

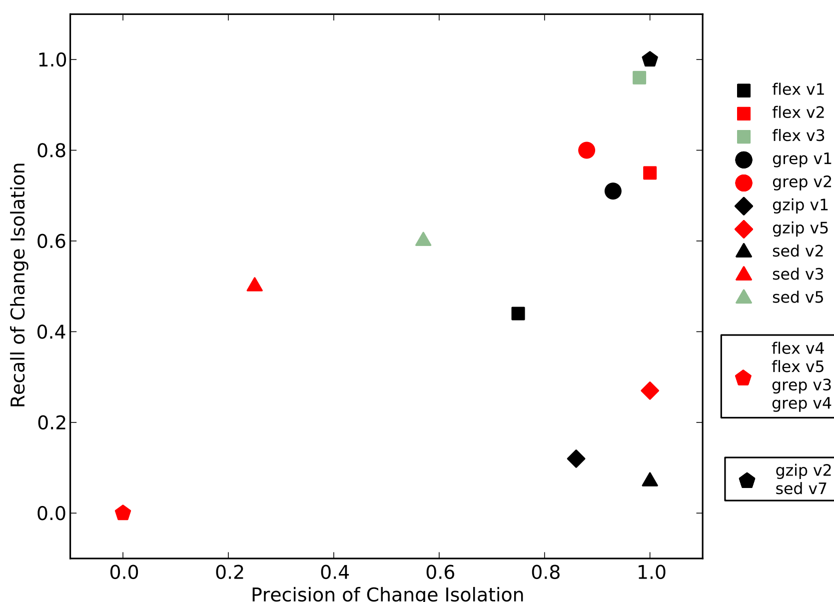


Figure 5.7: Precision and recall of associating failure-inducing changes with bug predictors

the iterative bug isolation algorithm described in chapter 2 (section 2.3.2) with c_i. We then enable all changes and collect feedback reports from this composite version. We calculate the conditional coverage profile from the different changes using branch profiles obtained from the feedback reports. We apply our change-isolation technique mentioned above to associate each bug predictor for the composite version with a change. (We find that the list of bug predictors found in the composite version includes only the bug predictors found in the versions with each individual change enabled). We evaluate this association by comparing it to the ground truth. As mentioned earlier, there are two criteria in this evaluation. First is the precision of our association: the fraction of predictors that are associated with the correct change. Since we do not associate any change when there are ties in the conditional coverage values, we also evaluate the recall: the fraction of total predictors that are associated with some change (irrespective of whether it is right or wrong). Figure 5.7 shows a scatter plot of the precision and recall of our technique. Each point in the plot corresponds to one test subject, except for two groups of overlapping points at $(0,0)$ and $(1,1)$.

Our technique has a precision of at least 50% for eleven of the sixteen test subjects. For ten of these subjects, the precision is at least 75%. Recall is 0 (and precision is undefined) for four test subjects. They

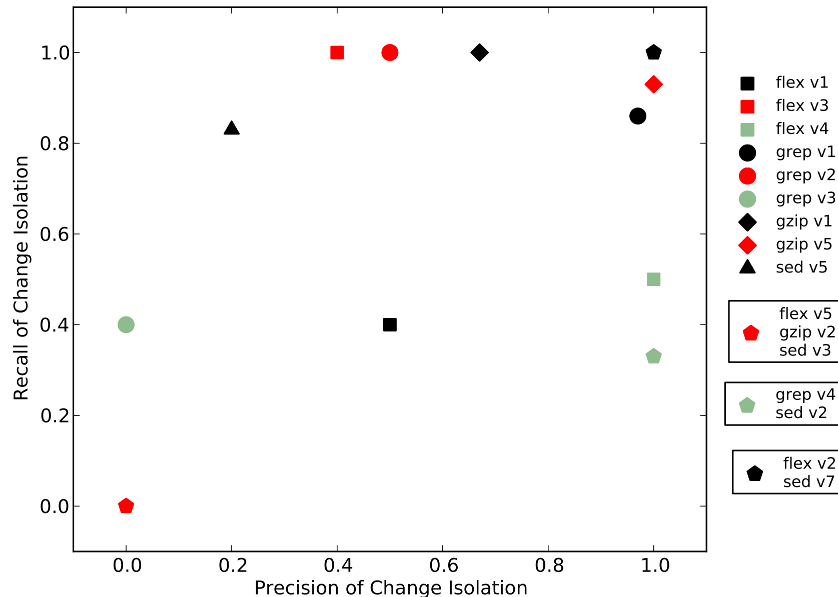


Figure 5.8: Precision and recall of associating failure-inducing changes with failing locations

are shown at $(0,0)$ in the scatter plot. The recall of our technique varies from 0% to 100%, with an average of 55%. The common symptom in the cases having low recall and/or precision is the same as the reason for ties in the values of conditional coverage. Either the atomic changes are indistinguishably close to each other or the predictors are sufficiently close to `main` to have tied conditional coverage values from multiple changes or a higher value in the conditional coverage from an irrelevant change. However, the average precision of our technique is 89%. For the proposed usage scenario, this high precision is preferable rather than increasing recall at the cost of precision.

5.3.2 Associating Changes with Failing Locations

Crash stacks, or just the crashing locations, may sometimes be available as part of the feedback reports. CBI uses only the labels (success or failure) associated with feedback reports for statistical analysis. However, when failed locations are available, we can use equation (5.12) to find the change that is most likely to impact these locations. In our experiments, multiple test cases may fail at the same location. Such failing locations are counted just once in our evaluation.

Since most of our test programs fail by producing incorrect output, we use the output tracer used in section 4.4 to find the first location where a wrong byte is output. The ground truth is obtained by enabling each change individually, and finding the failing locations. Failures at these locations in the composite version are attributed to this change. We run the composite version with output tracing to find failing locations. (Again, we find that the set of failing locations of the composite version is the same as the union of the failing locations found in the versions with each individual change enabled). We then use equation (5.12) to associate a change with each such location.

Figure 5.8 plots the precision and recall of this classification. Each point corresponds to one test subject, except for three groups of overlapping points at $(0, 0)$, $(1, 0.33)$, and $(1, 1)$. This classification has a precision of at least 50% for ten of the sixteen subjects. For seven of these subjects, the precision is at least 90%. The cases having low recall and precision reveal an interesting dual to the situations encountered in the previous subsection. Most of the failing locations that are not classified or are wrongly classified are located in utility routines that print the program's output. Such functions, because they are called from several locations, have high values in the conditional coverage profiles of several atomic changes. This leads to to tied values or incorrect classification. This is similar to the pathological cases in the previous section, where predictors closer to `main` are not classified or had incorrect classifications.

The average precision and recall of this classification are 88% and 78%, respectively. This average recall and precision are not close to the centroid of the points in fig. 5.8. This is because each subject has different numbers of failing locations. The average precision and recall mentioned above are overall averages rather than an average of averages.

5.3.3 Association using Forward Slices

Forward slicing computes a safe over-approximation of the set of program nodes that may be impacted by a changed node. A forward slice can be used to perform a classification similar to equation (5.12). A node is either present in the forward slice from a changed node or not present. If a bug predictor or a failing location is in the forward slice from exactly one change, we can associate the bug predictor or failing location with that change. If a bug predictor or a failing location is in the forward slice from multiple

changes, we cannot decisively make a association.

The advantage of using forward slicing is the absence of false negatives in the result of impact analysis. Hence, if a bug predictor or a failing location is associated with a change, that association is likely to be correct. The disadvantage of using slicing is that slices can be very large (up to 62% [BH03]), reducing the likelihood that a bug predictor or failing location is in the forward slice of just one change. This is confirmed by our experiments. For our test subjects, we used impact-analysis results from forward slicing to associate bug predictors and failing locations with changes. The precision of this classification is 100%. However, a majority of the bug predictors and failing locations are in the forward slices from multiple changes, and hence are not associated with any change. The recall is 3% for bug predictors and 1% for classifying failing locations. As opposed to the Boolean likelihood of impact computed using forward slicing, the fine-grained likelihood of impact computed using conditional coverage profiles achieves a relatively high recall and a reasonable precision. Moreover, our technique for computing conditional coverage profiles uses only the control-flow graph of the program. Hence, it can scale up to larger programs, with hundreds of thousands of lines of code. In contrast, program slicing tools construct program-dependence graphs of the program; such tools are not likely to scale up well to such large programs.

5.4 Summary

In this chapter, we have presented a technique to identify which of the set of changes made to a program is most relevant to bugs. Our technique associates each bug predictor and each failing location with a unique change. This association is based on a technique for approximating the potential impact of each change on each program point. We define a *conditional coverage profile* and develop a technique to estimate it from branch and call-graph profiles. We have built a tool that finds the most likely failure-inducing change by combining the conditional coverage profile with the output of CBI's statistical analysis. Our evaluation was performed in sixteen experimental settings across four test programs. Our technique for associating bug predictors with changes has an average precision of 89% and an average recall of 55%. Our technique for associating a program's failing locations with a change has an average precision of 88% and an average recall of 78%. While we demonstrate our results using existing benchmark programs for fault isolation,

evaluation in an actual software deployment scenario can help us to further validate our approach and to promote widespread adoption.

Chapter 6

Related Work

We begin our discussion of related work with a survey of statistical debugging techniques in section 6.1. Subsequent sections explain the relevance of compound predicates (section 6.2), adaptive bug isolation (section 6.3), and conditional coverage profiles (section 6.4) to related work. These sections also mention related work specific to the topic under discussion.

6.1 Survey of Statistical Debugging Techniques

This section briefly surveys related work on statistical debugging, both from the CBI project as well as other research groups.

6.1.1 Extensions from the CBI Project

In addition to the statistical debugging algorithm presented in section 2.3, two new analyses that use machine learning have been developed by the CBI project. The feedback reports collected by CBI are especially suited for applying machine learning algorithms. The data analyzed by many machine learning algorithms follow the *document-word* model. The data is modeled as a list of *documents*, and each document as a collection of *words*. Typical tasks performed on this data include classification, clustering and topic modeling. In the context of CBI, each feedback report is a document and predicates with nonzero counts are the words in the document.

Zheng et al. [Zhe+06] develop a bi-clustering algorithm to cluster runs failing under similar circumstances. Predicates are clustered together based on the failed runs in which they are observed. Andrzejewski et al. [And+07] propose ΔLDA as an extension to the machine learning technique of Latent Dirichlet Allocation (LDA). LDA recognizes the fact that documents may correspond to multiple *topics* rather than just one. In the context of CBI, the runtime behavior of a program may correspond to multiple usage scenarios (topics). ΔLDA considers two kinds of topics: *bug* topics that are found only in failed runs and *usage* topics that can be found in both successful and failed runs. In our experience, the clusters of predicates produced by these approaches is hard to evaluate subjectively and objectively. The small list of ranked predicates produced by the iterative elimination algorithm (section 2.3) is much more suitable for this task. Hence, we used it for evaluation in the earlier chapters.

BTrace is a static analysis tool that extends the set of bug predictors found by statistical analyses. It finds the shortest control-flow- and dataflow- feasible path in the program that visits all of the bug predictors. This analysis allows a programmer to examine the failure-predicting behavior even if the connection to a bug is not easily identifiable, or if the predictors are numerous or complex enough to overwhelm a programmer examining them directly.

6.1.2 Fault Localization Tools

SOBER [Liu+05] is a statistical debugging technique that uses divergence between a predicate's behavior in successful and failed runs as a measure of failure predictivity. The *evaluation bias* of a predicate in a run is the fraction of times it evaluated to true. If a predicate is never evaluated during a run, its evaluation bias is set to 0.5. A predicate's behavior is defined as the probability density function of its evaluation bias across all feedback reports. Unlike the technique presented in section 2.3, which only considers whether a predicate was ever true in a run, the definition of evaluation bias takes into account the actual number of times each predicate was evaluated in the run. This work does not perform any experimental evaluation with sampling enabled. In the presence of sampling, there will be numerous predicates that are never evaluated. Thus, the probability density function of the evaluation bias will have a spike at 0.5 in both the successful and failed runs. The effect of this scenario in their bug-relevance score is not studied.

The Gamma project represents one of the first practical systems for run-time monitoring of deployed software. Orso et al. [Ors+02] describe a data collection infrastructure, termed *software tomography*, that supports a variety of software evolution tasks and allows post-deployment changes to data collection. However, they rely on statically selective sampling for maintaining low overheads. Visualization techniques developed by the Gamma group [Jon+02; Ors+04] can prove useful to help programmers understand and interpret the results of fault localization. Tarantula [JH05] is a fault localization technique that uses statement coverage as predicates and weighted failure rate as the scoring metric.

The Holmes project [Chi+09] makes two orthogonal contributions to statistical debugging. The first is a *path* instrumentation scheme that counts the number of times each path is taken in an acyclic region. They find that path predicates are usually better failure predictors than branch and scalar-pairs predicates. However, it is costly to collect path predicates. Their second contribution is a form of adaptive predicate selection, similar to the idea explored in chapter 4, but at the granularity of functions. Based on partial feedback data, weak predictors are selected. During the adaptation step, functions close to the weak predictors in the program-dependence graph are chosen and predicates in these functions are instrumented during the next iteration. Holmes also strengthens weak predictors by selecting path predicates in functions containing weak branch predicates. A detailed comparison between adaptive bug isolation and Holmes was presented in section 4.4.5.

Gore et al. [Gor+11] use *elastic* predicates that use a more fine-grained decomposition of a *site* into *predicates*. For the returns and scalar-pairs schemes, which compare two scalar values, elastic predicates record the magnitude of difference in addition to the relation between the values. For example, consider a returns site. If μ_r and σ_r are respectively the mean and standard deviation of the value r returned at the site, several predicates of the form

$$(\mu_r + 3\sigma_r) \leq r$$
$$(\mu_r + 2\sigma_r) \leq r < (\mu_r + 3\sigma_r)$$

etc. are added. Data collection happens in two steps. The first step adds instrumentation to estimate the values of μ_r and σ_r for the various sites. In the second step, the program is instrumented to collect the

fine-grained predicates mentioned above. This instrumented version is used to gather feedback reports. Gore et al. [Gor+11] and Chilimbi et al. [Chi+09] use the *Importance* score defined in section 2.3 to perform fault isolation.

Except for the coarse-grained adaptivity explored by Holmes, none of the above related work directly explores the ideas proposed in this dissertation. Our contributions are mostly orthogonal, and can be integrated with the related work mentioned here.

6.2 Related Work for Compound Predicates

The idea of propositional combinations of simple failure predictors can be incorporated into other fault localization tools. Techniques that also use the *Importance* score, namely path predicates, and elastic predicates, are especially suited, since the upper-bounds estimated in section 3.1.2 are directly applicable.

Tarantula uses statement-level coverage as candidate failure predictors. For Tarantula, co-execution of statements is the extension equivalent to compound predicates. The general idea of pruning is also applicable in this scenario. The estimates on the bounds must be reworked for the scoring metrics used by Tarantula. The *effort* metric described in section 3.2 is still applicable.

The data analysed by SOBER is a probability vector, with each value representing the estimated chance of a simple predicate being true when observed. The similarity in collected data means that similar techniques for complex predicate generation are applicable. The three-valued logic described in subsection 3.1.1 could be replaced with joint-probability when generating conjunctions; De Morgan's law can be applied to generate disjunctions. As with Tarantula, the idea of pruning can be applied by reworking the upper-bounds on the score of a compound predicate.

Compound predicates relate behavior at multiple program points, and therefore may be difficult to comprehend. Presenting compound predicates in a way that programmers can readily understand remains an open problem. The Gammatella [Ors+04] project explores visualization of program-execution data, such as failure data. Their ideas may be applicable for visualizing compound predicates.

Haran et al. [Har+07] analyze data from deployed software to classify executions as *success* or *failure*. They use tree-based classifiers and association rules to model "failure signals". The interior nodes in

such trees are simple predicates, and the leaf nodes are the predicted outcome. Tree-based classifiers and association rules implicitly encode conjunctions and disjunctions. The goal of their technique is classification of executions rather than fault localization.

Daikon [Ern+01] detects invariants in a program by observing multiple program runs. Invariants are predicates generated using operators such as sum and max to combine program variables and collection (e.g., array) objects. Daikon is intended for many uses beyond bug isolation, and so it monitors a much larger set of predicates than CBI. This makes scalable complex predicate generation more difficult. However, Dodoo et al. [Dod+02] have successfully extended the work to generate implications from the simpler, measured predicates. Dodoo et al. alternate clustering and invariant detection to find invariant implications over a set of program runs. The initial clustering is performed using the *k*-means algorithm [Jai+99], with program runs represented as normalized vectors of scalar variable values. Since CBI represents run information as bit-vectors this technique can be applied essentially unchanged. However, CBI's focus is fault localization under sparse sampling conditions. Predicates will rarely be identified as invariant in the presence of sampling.

6.3 Related Work on Adaptive Instrumentation

To the best of our knowledge, Holmes is the only tool that uses adaptation for purposes of post-deployment fault localization. Several dynamic program analyses developed for other purposes alter their behavior adaptively. The dynamic leak detector of Hauswirth and Chilimbi [HC04] profiles code segments at a rate inversely proportional to their execution frequencies. Yu et al. [Yu+05] use dynamic feedback to control the granularity of locksets and threadsets in their data race detection algorithm. Dwyer et al. [Dwy+07] make adaptive, online decisions to monitor just a subset of the program events in their dynamic finite-state property verifier. The AjaxScope platform for monitoring client side execution of web applications [KL07] provides mechanisms for specifying adaptive policies; the authors describe a performance profiling tool using this feature.

The Paradyn project [Mil+95] uses adaptive, dynamic instrumentation for performance profiling of large parallel programs. Roth and Miller [RM06] emphasize automated, on-line diagnosis of performance

bottlenecks. Unlike Paradyn and the other online adaptive analyses [Dwy+07; HC04; KL07; Yu+05] our approach uses statistical bug detection with data being aggregated across many runs and analysis being performed offline. Paradyn's tools and techniques for managing and visualizing large data streams may be useful in our domain as well.

In their execution classification tool mentioned in section 6.2, Haran et al. [Har+07] select program behaviors to be monitored using weighted sampling. In-house data collection finds predicates that are useful for classification, and sampling is biased towards such predicates. Like CBI, sampling is used to reduce monitoring overhead. In the presence of a large user community, weighted sampling can be combined with our technique for a non-uniform assignment of instrumentation sites to users using any of our heuristics as weights.

Software tomography, the data-collection infrastructure developed by the Gamma project, statically divides the data-collection tasks and distributes them among the user community. However, they assume that the selection of those tasks is human-directed. The adaptive bug isolation technique we propose is an automatic, heuristically-guided system for bug-hunting that changes data-collection tasks in response to feedback. However, we do assume that the programmer is watching the results of fault isolation, and can choose to continue the adaptive search, or stop the process if he recognizes and can fix the bug.

Renieris and Reiss [RR03] present a model of the debugging activity performed by the programmer. They posit that when given a predictor of failure, the programmer searches for clues about the failure by performing an undirected breadth-first search on the program-dependence graph. This model has been used by statistical debugging tools [CZ05; JH05; JS07; Liu+05; Zhe+06] to quantify the effectiveness of the bug predictors found by their techniques. In chapter 3 (section 3.2), we use this model to develop the *effort* metric that characterizes the usability of a compound predicate to a programmer. In chapter 4, we adopt a model similar to Renieris and Reiss [RR03], but use a heuristic search instead of breadth-first search. It should be noted that in earlier work, the model was used as an independent metric to compare different analyses. Here, we use it for a different goal: to rank compound predicates based on their utility, and to reduce the monitoring overhead.

6.4 Related Work for Conditional Coverage Profiles

The fault localization tools mentioned in section 6.1 present a list of program behavior (predicates, statements etc.) that are symptomatic of failure. They do not explore the idea of extending this information to find failure-inducing changes. We believe that this additional information is useful in helping programmers understand bugs and develop fixes. Potentially, the classifier described in section 5.3 can combine results from any of these tools with conditional-coverage profiles to help identify failure-inducing changes. In this section, we discuss related work for the ideas in chapter 5 across three topics: change impact analyses, tools that identify failure-inducing changes, and probabilistic static analyses.

6.4.1 Change Impact Analyses

Change impact analysis identifies the components that are potentially impacted by a change to a program. Prior research has tackled change impact analysis using static and dynamic analyses. These techniques find the set of impacted components at various granularities, depending on how the results are expected to be used. Impact sets can be tracked in coarser granularities at the procedure and object level, or finer granularities like basic blocks or source lines. Static techniques that track impact sets at coarser granularities perform a closure of the call-graph [BA96] or object-relationship graph [Kun+94]. Static slicing is used to track impact sets at the granularity of program-dependence graph nodes [Bin97; Hor+90; Wei81]. Dynamic techniques like PathImpact [LR03b], Execute-After Sequences [Api+05], Incremental PathImpact [LR03a], and Chianti [Ren+04] use the dynamic trace of method calls to prune the call-graph closure obtained from static techniques. Dynamic slicing techniques [KL88; Zha+06] can be used to find impact sets corresponding to a particular execution. CoverageImpact [Ors+03] intersects compacted coverage information and static slices to perform impact analysis aggregated over several executions.

For dynamic or hybrid analyses, the granularity of their results affects the run-time overhead for data collection. Since dynamic techniques rely on information gathered from a set of executions, the precision of their results is not guaranteed. Techniques that use lightweight data collection are amenable to the post-deployment setting where a wider range of operational profiles can be collected. To our knowledge, the CoverageImpact technique proposed by the Gamma project [Ors+03] is the only one designed to

use data from post-deployment monitoring. However, like other impact analysis techniques, it assumes precise coverage information and makes binary decisions on whether a statement is affected by a change. Conditional coverage profiles generalize this approach by computing probabilistic measures of impact. They are computed using the sparse random data collected using CBI's post-deployment monitoring techniques.

Static slicing is one approach to perform change impact analysis at the granularity of program-dependence graph nodes. A *forward slice* is computed from the changed location to identify the set of affected nodes. Program slicing is a more general tool, that has been used to solve other problems besides change impact analysis. Such applications include failure comprehension and debugging, regression testing [RH97], and program integration [Bin+95; San+10]. Recent techniques for probabilistic slicing [SH10; Sin06] explore the problem of quantifying the impact of a change by assigning a probabilistic value to each node in the forward slice. Singer [Sin06] proposes that the impact probability on a node be the product of the conditional probability on edges dominating the node. We are not aware of any implementation or evaluation of this proposal. Santelices and Harrold [SH10] break the effect of a change into three components: coverage probability, propagation probability, and impact probability. However, Santelices and Harrold do not use run-time profiles and assume that each outcome is equally likely at a branch statement. The novelty of our approach is that our formalization, the conditional coverage profile, is a concrete property of program executions, giving us an ideal target. We also incorporate run-time profiles that are efficiently collected using post-deployment monitoring.

6.4.2 Identifying Failure-inducing Changes

The problem of identifying failure-inducing changes has been studied earlier. Zeller [Zel99] uses the delta debugging algorithm to identify a minimal set of atomic changes that cause failure when applied. In this technique, subsets of the changes are enabled and the program is tested for failure. Subsets that do not cause failure are eliminated. This divide-and-conquer step is repeatedly applied to find the minimal set of changes that cause failure. Due to relations between changes, building intermediate versions with arbitrary subsets of changes enabled might generate compiler errors. Ren and Ryder [RR07] and Zhang

et al. [Zha+08] use static and dynamic information to select more relevant subsets of changes to enable while finding the minimal set of failure-inducing changes. However, these works all assume the ability to enable each change individually. Consequently, they are best suited for in-house testing, where it is possible to discover failures and collect profiles with a subset of changes enabled.

Hoffman et al. [Hof+09] and Störzer et al. [Stö+06] propose techniques to identify failure-inducing changes without constructing any intermediate versions. A critical piece of information used by these techniques is whether a change affects a failed test, i.e., whether a changed location is executed in the failed test. Their algorithms are not robust to incompleteness in this information, and hence are not viable in the presence of sampling. On the other hand, the goal of our work, and other existing work on statistical debugging, is to target bugs that evade in-house testing and cause problems post-deployment. The use of sampling and the inability to build intermediate versions (with subsets of changes enabled) are critical constraints in this domain, and set our technique apart from other existing work.

6.4.3 Probabilistic Static Analyses

More generally, runtime profiles have been used to quantify outputs of static analyses. Probabilistic pointer analyses [Che+04; DSS06] associate probabilities to points-to and alias relations. Dataflow frequency analysis [MS01; Ram96] uses runtime profiles to compute the frequency or probability with which the dataflow facts hold true. Their framework is applicable to analyses such as reaching definitions, available expressions, and live variables. The problem of computing conditional coverage profiles can be considered as an instance of probabilistic dataflow analysis. The associated dataflow problem is the computation of reachability from a specific source node. Some aspects of our approach, such as the use of symbolic expressions, and the use of runtime profiles are similar to this work. The differences include our use of call-graph profiles, and the usage of mutual exclusion between the predecessors of a node. Ramalingam [Ram96] handle procedure calls by computing summary information for each procedure. Since we are estimating the probability of execution after a chosen source node for the entire program, call-graph profiles are useful to us.

Chapter 7

Conclusion

Prior work has established post-deployment statistical debugging as a viable and useful testing methodology. This dissertation develops various techniques to improve the monitoring efficiency and fault localization of the state-of-the-art in statistical debugging. Adaptive bug isolation sacrifices monitoring of the whole program in favor of selective, low-overhead instrumentation. Compound predicates improve fault localization by finding better predictors for complex bugs, while imposing no additional monitoring overhead. Conditional coverage profiles augment the output of statistical debugging with source-level changes that are likely causes of failures. In addition to their relevance to CBI, conditional coverage profiles can potentially be used to extend existing tools that visualize impacted sets. It can also lead us to revisit regression test prioritization, which is another application of impact analysis.

Our contributions, while seemingly orthogonal in their goals, are unified by their application of static analysis to connect dynamic behavior. Program dependences are used to prune less useful compound predicates, thus making statistical debugging with compound predicates tractable. Control dependences are used to selectively instrument those predicates that are likely to be good predictors of failure. Conditional coverage profiles, derived from control-flow graphs and runtime profiles, are used to associate bug predictors with failure-inducing changes.

Our incorporation of static analysis into CBI is one step towards realizing the potential of static analysis driving dynamic analysis, and vice versa, envisioned by Liblit in his “Reflections on the Role of

Static Analysis in Cooperative Bug Isolation" [Lib08]. Static program structure is used to drive dynamic analysis in adaptive bug isolation. Program structure is also used to quantify the usability of compound predicates. Alternatively, dynamic program behavior in the form of runtime profiles is used to augment static change impact analysis. Liblit is also skeptical about the scalability and robustness of static analyses. For our purposes of constructing program-dependence graphs and precise pointer analysis, the CodeSurfer [Gra06] tool was able to scale up to programs with upwards of 50KLOC. The program-dependence graphs and points-to sets might be imprecise in the presence of buffer overflows and wild pointers. Such imprecision can negatively impact our techniques. But the benefits in terms of low overheads and better fault localization outweigh such negative impact.

7.1 Interoperability of the Contributions

In chapter 1, we mentioned the twin goals in the design of CBI: gathering meaningful data for effective fault localization, and reducing the overhead of monitoring to facilitate post-deployment data collection. Adaptive bug isolation focuses on reducing runtime overheads, while the other two contributions focus on improving fault localization. The compatibility of these techniques is worth further consideration.

A compound predicate is useful for fault localization only when its component predicates are simultaneously instrumented. Adaptive bug isolation, on the other hand, omits monitoring of a vast majority of predicates. The components of most compound predicates will not be instrumented simultaneously. Thus, adaptive bug isolation drastically reduces the number of compound predicates available for statistical analysis.

Identification of failure-inducing changes, on the other hand, is compatible with the other two contributions. Conditional coverage profiles make no assumption on how the branch and call-graph profiles are collected. When used with adaptive bug isolation, profiles for some branch and call-graph edges might not be available. Uniform division at the sources of such edges can be assumed, until adaptive predicate selection chooses to instrument them. Conditional coverage profiles can also be used to associate a failure-inducing change with compound predicates. A potential classification heuristic would be choosing the change that is most likely to impact both the components of a compound predicate.

7.2 Closing Thoughts

Statistical debugging brings together several research areas, including static analysis, dynamic analysis, statistics, and machine learning. This dissertation draws upon some of these areas. Some ideas illustrate how ideas in these different fields complement each other. Dynamic analysis augments static change impact analysis. Machine learning, as shown by ΔLDA, improves dynamic analysis of the runtime behavior of predicates. Static analysis helps removal of redundant instrumentation.

The converse is also true: seemingly obvious enhancements in one domain fail due to interactions with other domains. While finding failure-inducing changes, clustering-based classifiers were only marginally better than a simple classifier (section 5.3). The Mann–Whitney U-test does not assume normal distribution of data and, hence, is more generally applicable than the t-test. However, it performed worse than the t-test during adaptive bug isolation (chapter 4).

The potential of the use of static analysis has not been fully realized yet. Deeper static analysis, in the form of model checking and theorem proving, can be used to prove relations between CBI's predicates. Such relations can avoid redundant instrumentation and can be encoded as domain knowledge in machine learning algorithms. In our experiments, none of the model checkers we tried scaled up beyond tiny programs. However, the potential benefits make further investigation worthwhile.

Overall, we have enhanced the appeal of statistical debugging to both programmers and end-users. We hope this brings statistical debugging closer to the mainstream, leading to fewer bugs and happier users.

Bibliography

[And+07] David Andrzejewski, Anne Mulhern, Ben Liblit, and Xiaojin Zhu. "Statistical Debugging Using Latent Topic Models". In: *ECML*. Ed. by Joost N. Kok, Jacek Koronacki, Ramon López de Mántaras, Stan Matwin, Dunja Mladenic, and Andrzej Skowron. Vol. 4701. Lecture Notes in Computer Science. Springer, 2007, pp. 6–17.

[Api+05] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. "Efficient and precise dynamic impact analysis using execute-after sequences". In: *ICSE '05: Proceedings of the 27th international conference on Software engineering*. St. Louis, MO, USA: ACM, 2005, pp. 432–441.

[AR01] Matthew Arnold and Barbara G. Ryder. "A Framework for Reducing the Cost of Instrumented Code". In: *PLDI*. 2001, pp. 168–179.

[Arn96] Robert S. Arnold. *Software Change Impact Analysis*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1996.

[BA96] Shawn Bohner and Robert Arnold. *An introduction to software change impact analysis*. Ed. by Shawn Bohner and Robert Arnold. IEEE Computer Society Press, 1996, pp. 1–26.

[Bal+05] Gogul Balakrishnan, Radu Gruian, Thomas W. Reps, and Tim Teitelbaum. "CodeSurfer/x86–A Platform for Analyzing x86 Executables". In: *CC*. Ed. by Rastislav Bodík. Vol. 3443. Lecture Notes in Computer Science. Springer, 2005, pp. 250–254.

[BH00] Bryan Buck and Jeffrey K. Hollingsworth. "An API for Runtime Code Patching". In: *Int. J. High Perform. Comput. Appl.* 14.4 (2000), pp. 317–329.

[BH03] D. Binkley and M. Harman. "A large-scale empirical study of forward and backward static slice size and context sensitivity". In: *Proceedings of the 2003 International Conference on Software Maintenance*. Amsterdam, The Netherlands: IEEE Computer Society, Sept. 2003.

[Bin+95] David Binkley, Susan Horwitz, and Thomas Reps. "Program integration for languages with procedure calls". In: *ACM Trans. Softw. Eng. Methodol.* 4.1 (Jan. 1995), pp. 3–35. url: `http://doi.acm.org/10.1145/201055.201056`.

[Bin97] David Binkley. "Semantics Guided Regression Test Cost Reduction". In: *IEEE Trans. Softw. Eng.* 23.8 (1997), pp. 498–516.

[BL96] Thomas Ball and James R. Larus. "Efficient Path Profiling". In: *MICRO*. 1996, pp. 46–57.

[BR07] Gogul Balakrishnan and Thomas Reps. "DIVINE: discovering variables in executables". In: *Proceedings of the 8th international conference on Verification, model checking, and abstract interpretation*. VMCAI'07. Nice, France: Springer-Verlag, 2007, pp. 1–28. url: `http://dl.acm.org/citation.cfm?id=1763048.1763050`.

[CE04] Kalyan-Ram Chilakamarri and Sebastian G. Elbaum. "Reducing Coverage Collection Overhead With Disposable Instrumentation". In: *ISSRE*. IEEE Computer Society, 2004, pp. 233–244.

[Che+04] Peng-Sheng Chen, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee. "Interprocedural Probabilistic Pointer Analysis". In: *IEEE Trans. Parallel Distrib. Syst.* 15.10 (Oct. 2004), pp. 893–907.

[Chi+09] Trishul M. Chilimbi, Ben Liblit, Krishna K. Mehra, Aditya V. Nori, and Kapil Vaswani. "Holmes: Effective statistical debugging via efficient path profiling". In: *ICSE*. IEEE, 2009, pp. 34–44.

[CZ05] Holger Cleve and Andreas Zeller. "Locating causes of program failures". In: *ICSE*. Ed. by Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh. ACM, 2005, pp. 342–351.

[Dod+02] Nii Dodoo, Alan Donovan, Lee Lin, and Michael D. Ernst. *Selecting Predicates for Implications in Program Analysis*. Draft. `http://pag.csail.mit.edu/~mernst/pubs/invariants-implications.ps`. 2002.

[DSS06] Jeff Da Silva and J. Gregory Steffan. “A probabilistic pointer analysis for speculative optimizations”. In: *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. ASPLOS-XII. San Jose, California, USA: ACM, 2006, pp. 416–425. url: `http://doi.acm.org/10.1145/1168857.1168908`.

[Dwy+07] Matthew B. Dwyer, Alex Kinneer, and Sebastian G. Elbaum. “Adaptive Online Program Analysis”. In: *ICSE*. IEEE Computer Society, 2007, pp. 220–229.

[Ern+01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. “Dynamically discovering likely program invariants to support program evolution”. In: *IEEE Transactions on Software Engineering* 27.2 (Feb. 2001). A previous version appeared in *ICSE ’99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999, pp. 99–123.

[Fer+87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. “The Program Dependence Graph and Its Use in Optimization”. In: *ACM Trans. Program. Lang. Syst.* 9.3 (1987), pp. 319–349.

[For07] Mike Fortin (Distinguished Engineer, Windows Core Operating Systems Division, Microsoft Corporation). *Limiting executable footprints*. Personal communication. Nov. 2007.

[Fre] Free Software Foundation. *GCC: The GNU compiler collection*. `http://gcc.gnu.org/`.

[Gor+11] Ross Gore, Paul F. Reynolds, and David Kamensky. “Statistical debugging with elastic predicates”. In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 492–495. url: `http://dx.doi.org/10.1109/ASE.2011.6100107`.

[Gra06] GrammaTech. *CodeSurfer*. `http://www.codesurfer.com`. Sept. 2006.

[Har+07] Murali Haran, Alan F. Karr, Michael Last, Alessandro Orso, Adam A. Porter, Ashish P. Sanil, and Sandro Fouche. “Techniques for Classifying Executions of Deployed Software to Support Software Engineering Tasks”. In: *IEEE Trans. Software Eng.* 33.5 (2007), pp. 287–304.

[HC04] Matthias Hauswirth and Trishul M. Chilimbi. “Low-overhead memory leak detection using adaptive statistical profiling”. In: *ASPLOS*. Ed. by Shubu Mukherjee and Kathryn S. McKinley. ACM, 2004, pp. 156–164.

[Hof+09] Kevin J. Hoffman, Patrick Eugster, and Suresh Jagannathan. “Semantics-aware trace analysis”. In: *PLDI*. Ed. by Michael Hind and Amer Diwan. ACM, 2009, pp. 453–464.

[Hor+10] Susan Horwitz, Ben Liblit, and Marina Polishchuk. “Better Debugging via Output Tracing and Callstack-Sensitive Slicing”. In: *IEEE Transactions on Software Engineering* 36.1 (2010), pp. 7–19.

[Hor+88] S. Horwitz, T. Reps, and D. Binkley. “Interprocedural slicing using dependence graphs”. In: *SIGPLAN Not.* 23.7 (June 1988), pp. 35–46. url: `http://doi.acm.org/10.1145/960116.53994`.

[Hor+90] S. Horwitz, T. Reps, and D. Binkley. “Interprocedural slicing using dependence graphs”. In: *ACM Transactions on Programmings Languages and Systems* 12.1 (Jan. 1990), pp. 26–60.

[HR92] Susan Horwitz and Thomas W. Reps. “The Use of Program Dependence Graphs in Software Engineering”. In: *ICSE*. 1992, pp. 392–411.

[Hut+94] Monica Hutchins, Herbert Foster, Tarak Goradia, and Thomas J. Ostrand. “Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria”. In: *ICSE*. 1994, pp. 191–200.

[Jai+99] A. K. Jain, M. N. Murty, and P. J. Flynn. “Data Clustering: A Review”. In: *ACM Computing Surveys* 31.3 (Sept. 1999), pp. 264–323. url: `citeseer.ist.psu.edu/jain99data.html`.

[JH05] James A. Jones and Mary Jean Harrold. “Empirical evaluation of the Tarantula automatic fault-localization technique”. In: *ASE*. Ed. by David F. Redmiles, Thomas Ellman, and Andrea Zisman. ACM, 2005, pp. 273–282.

[Jon+02] James A. Jones, Mary Jean Harrold, and John T. Stasko. "Visualization of test information to assist fault localization". In: *ICSE*. ACM, 2002, pp. 467–477.

[JR94] Daniel Jackson and Eugene J. Rollins. *Chopping: A Generalization of Slicing*. Tech. rep. Pittsburgh, PA, USA, 1994.

[JS07] Lingxiao Jiang and Zhendong Su. "Context-aware statistical debugging: From bug predictors to faulty control flow paths". In: *ASE*. Ed. by R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer. ACM, 2007, pp. 184–193.

[KL07] Emre Kiciman and V. Benjamin Livshits. "AjaxScope: a platform for remotely monitoring the client-side behavior of web 2.0 applications". In: *SOSP*. Ed. by Thomas C. Bressoud and M. Frans Kaashoek. ACM, 2007, pp. 17–30.

[KL88] Bogdan Korel and Janusz W. Laski. "Dynamic Program Slicing". In: *Inf. Process. Lett.* 29.3 (1988), pp. 155–163.

[Kun+94] David Chenho Kung, Jerry Gao, Pei Hsia, F. Wen, Yasufumi Toyoshima, and Cris Chen. "Change Impact Identification in Object Oriented Software Maintenance". In: *ICSM*. Ed. by Hausi A. Müller and Mari Georges. IEEE Computer Society, 1994, pp. 202–211.

[Lal+06] Akash Lal, Junghee Lim, Marina Polishchuk, and Ben Liblit. "Path Optimization in Programs and its Application to Debugging". In: *15th European Symposium on Programming*. Ed. by Peter Sestoft. Vienna, Austria: Springer, Mar. 2006, pp. 246–263.

[Lev69] David A. Levine. "Algorithm 344: Student's t-distribution [S14]". In: *Commun. ACM* 12.1 (1969), pp. 37–38.

[Liba] Ben Liblit. *CBI Documentation: Compiling Instrumented Executables*. `http://research.cs.wisc.edu/cbi/developers/guide/ar01s02.html`.

[Libb] Ben Liblit. *The Cooperative Bug Isolation Project*. `http://www.cs.wisc.edu/cbi/`.

[Lib+03] Ben Liblit, Alexander Aiken, Alice X. Zheng, and Michael I. Jordan. "Bug isolation via remote program sampling". In: *PLDI*. ACM, 2003, pp. 141–154.

[Lib+05] Ben Liblit, Mayur Naik, Alice X. Zheng, Alexander Aiken, and Michael I. Jordan. "Scalable statistical bug isolation". In: *PLDI*. Ed. by Vivek Sarkar and Mary W. Hall. ACM, 2005, pp. 15–26.

[Lib07] Ben Liblit. *Cooperative Bug Isolation (Winning Thesis of the 2005 ACM Doctoral Dissertation Competition)*. Vol. 4440. Lecture Notes in Computer Science. Springer, 2007.

[Lib08] Ben Liblit. "Reflections on the Role of Static Analysis in Cooperative Bug Isolation". In: *Proceedings of the 15th International Static Analysis Symposium*. European Association for Programming Languages and Systems. Valencia, Spain, July 2008, pp. 18–31.

[Liu+05] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. "SOBER: statistical model-based bug localization". In: *ESEC/SIGSOFT FSE*. Ed. by Michel Wermelinger and Harald Gall. ACM, 2005, pp. 286–295.

[LR03a] James Law and Gregg Rothermel. "Incremental Dynamic Impact Analysis for Evolving Software Systems". In: *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, p. 430.

[LR03b] James Law and Gregg Rothermel. "Whole Program Path-Based Dynamic Impact Analysis". In: *ICSE*. IEEE Computer Society, 2003, pp. 308–318.

[LS05a] Zhenkai Liang and R. Sekar. "Automatic Generation of Buffer Overflow Attack Signatures: An Approach Based on Program Behavior Models". In: *ACSAC*. IEEE Computer Society, 2005, pp. 215–224.

[LS05b] Zhenkai Liang and R. Sekar. "Fast and automated generation of attack signatures: a basis for building self-protecting servers". In: *ACM Conference on Computer and Communications Security*. Ed. by Vijay Atluri, Catherine Meadows, and Ari Juels. ACM, 2005, pp. 213–222.

[Mil+95] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. "The Paradyn Parallel Performance Measurement Tool". In: *IEEE Computer* 28.11 (1995), pp. 37–46.

[Mis+05] Jonathan Misurda, James A. Clause, Juliya L. Reed, Bruce R. Childers, and Mary Lou Soffa. “Demand-driven structural testing with dynamic instrumentation”. In: *ICSE*. Ed. by Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh. ACM, 2005, pp. 156–165.

[MS01] Eduard Mehofer and Bernhard Scholz. “A Novel Probabilistic Data Flow Framework”. In: *Proceedings of the 10th International Conference on Compiler Construction*. London, UK, UK: Springer-Verlag, 2001, pp. 37–51.

[MW47] H. B. Mann and D. R. Whitney. “On a Test of Whether One of Two Random Variables is Stochastically Larger than the Other”. In: *The Annals of Mathematical Statistics* 18.1 (1947), pp. 50–60.

[NS05] James Newsome and Dawn Xiaodong Song. “Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software”. In: *NDSS*. The Internet Society, 2005.

[Ors+02] Alessandro Orso, Donglin Liang, Mary Jean Harrold, and Richard J. Lipton. “Gamma system: continuous evolution of software after deployment”. In: *ISSTA*. 2002, pp. 65–69.

[Ors+03] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. “Leveraging field data for impact analysis and regression testing”. In: *ESEC / SIGSOFT FSE*. ACM, 2003, pp. 128–137.

[Ors+04] Alessandro Orso, James A. Jones, Mary Jean Harrold, and John T. Stasko. “Gammatella: Visualization of Program-Execution Data for Deployed Software”. In: *ICSE*. IEEE Computer Society, 2004, pp. 699–700.

[PO11] 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. Toronto, Ontario, Canada: ACM, 2011, pp. 199–209. url: `http://doi.acm.org/10.1145/2001420.2001445`.

[Qin+07] Feng Qin, Joseph Tucek, Yuanyuan Zhou, and Jagadeesan Sundaresan. “Rx: Treating bugs as allergies–a safe method to survive software failures”. In: *ACM Trans. Comput. Syst.* 25.3 (2007).

[Ram96] G. Ramalingam. “Data flow frequency analysis”. In: *SIGPLAN Not.* 31.5 (May 1996), pp. 267–277.

[Rei+06] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. “BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML”. In: *OSDI*. USENIX Association, 2006, pp. 61–74.

[Ren+04] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. “Chianti: a tool for change impact analysis of java programs”. In: *OOPSLA*. Ed. by John M. Vlissides and Douglas C. Schmidt. ACM, 2004, pp. 432–448.

[RH97] Gregg Rothermel and Mary Jean Harrold. “A safe, efficient regression test selection technique”. In: *ACM Trans. Softw. Eng. Methodol.* 6.2 (Apr. 1997), pp. 173–210. url: `http://doi.acm.org/10.1145/248233.248262`.

[RM06] Philip C. Roth and Barton P. Miller. “On-line automated performance diagnosis on thousands of processes”. In: *PPOPP*. Ed. by Josep Torrellas and Siddhartha Chatterjee. ACM, 2006, pp. 69–80.

[Rot+06] Gregg Rothermel, Sebastian Elbaum, Alex Kinneer, and Hyunsook Do. *Software-artifact Intrastructure Repository*. `http://sir.unl.edu/portal/`. Sept. 2006.

[RR03] Manos Renieris and Steven P. Reiss. “Fault Localization With Nearest Neighbor Queries”. In: *ASE*. IEEE Computer Society, 2003, pp. 30–39.

[RR07] Xiaoxia Ren and Barbara G. Ryder. “Heuristic ranking of java program edits for fault localization”. In: *ISSTA*. Ed. by David S. Rosenblum and Sebastian G. Elbaum. ACM, 2007, pp. 239–249.

[RR95] Thomas Reps and Genevieve Rosay. "Precise interprocedural chopping". In: *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*. SIGSOFT '95. Washington, D.C., United States: ACM, 1995, pp. 41–52. url: http://doi.acm.org/10.1145/222124.222138.

[San+10] Raul Santelices, Mary Jean Harrold, and Alessandro Orso. "Precisely Detecting Runtime Change Interactions for Evolving Software". In: *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*. ICST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 429–438. url: http://dx.doi.org/10.1109/ICST.2010.29.

[SH10] Raul Santelices and Mary Jean Harrold. *Probabilistic Slicing for Predictive Impact Analysis*. Tech. rep. GIT-CERCS-10-10. Georgia Tech, Nov. 2010.

[Sin06] Jeremy Singer. "Towards Probabilistic Program Slicing". In: *Beyond Program Slicing*. Ed. by David W. Binkley, Mark Harman, and Jens Krinke. Dagstuhl Seminar Proceedings 05451. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. url: http://drops.dagstuhl.de/opus/volltexte/2006/485.

[Stö+06] Maximilian Störzer, Barbara G. Ryder, Xiaoxia Ren, and Frank Tip. "Finding failure-inducing changes in java programs using change classification". In: *SIGSOFT FSE*. Ed. by Michal Young and Premkumar T. Devanbu. ACM, 2006, pp. 57–68.

[TH05] Mustafa M. Tikir and Jeffrey K. Hollingsworth. "Efficient online computation of statement coverage". In: *Journal of Systems and Software* 78.2 (2005), pp. 146–165.

[Vas+07] Kapil Vaswani, Aditya V. Nori, and Trishul M. Chilimbi. "Preferential path profiling: compactly numbering interesting paths". In: *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '07. Nice, France: ACM, 2007, pp. 351–362. url: http://doi.acm.org/10.1145/1190216.1190268.

[Wan+04] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. “Shield: vulnerability-driven network filters for preventing known vulnerability exploits”. In: *SIGCOMM*. Ed. by Raj Yavatkar, Ellen W. Zegura, and Jennifer Rexford. ACM, 2004, pp. 193–204.

[Wei06] Eric W. Weisstein. “Boolean Function”. In: *MathWorld–A Wolfram Web Resource* (2006). `http://mathworld.wolfram.com/BooleanFunction.html`.

[Wei81] Mark Weiser. “Program Slicing”. In: *ICSE*. Ed. by Seymour Jeffrey and Leon G. Stucki. IEEE Computer Society, 1981, pp. 439–449.

[Yin+04] Annie T. T. Ying, Gail C. Murphy, Raymond T. Ng, and Mark Chu-Carroll. “Predicting Source Code Changes by Mining Change History”. In: *IEEE Trans. Software Eng.* 30.9 (2004), pp. 574–586.

[Yu+05] Yuan Yu, Tom Rodeheffer, and Wei Chen. “RaceTrack: efficient detection of data race conditions via adaptive tracking”. In: *SOSP*. Ed. by Andrew Herbert and Kenneth P. Birman. ACM, 2005, pp. 221–234.

[Zel99] Andreas Zeller. “Yesterday, my program worked. Today, it does not. Why?” In: *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*. ESEC/FSE-7. Toulouse, France: Springer-Verlag, 1999, pp. 253–267. url: `http://dx.doi.org/10.1145/318773.318946`.

[Zha+06] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. “Pruning dynamic slices with confidence”. In: *SIGPLAN Not.* 41.6 (2006), pp. 169–180.

[Zha+08] Sai Zhang, Yu Lin, Zhongxian Gu, and Jianjun Zhao. “Effective identification of failure-inducing changes: a hybrid approach”. In: *PASTE*. Ed. by Shriram Krishnamurthi and Michal Young. ACM, 2008, pp. 77–83.

[Zhe+06] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. “Statistical debugging: simultaneous identification of multiple bugs”. In: *ICML*. Ed. by William W. Cohen

and Andrew Moore. Vol. 148. ACM International Conference Proceeding Series. ACM, 2006, pp. 1105–1112.