UNIVERSITY OF CALIFORNIA
RIVERSIDE

Dynamic State Alteration Techniques for Automatically Locating Software Errors

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Dennis Bernard Jeffrey

August 2009

Dissertation Committee:

Dr. Rajiv Gupta, Chairperson
Dr. Gianfranco Ciardo
Dr. Iulian Neamtiu

Copyright by
Dennis Bernard Jeffrey
2009

The Dissertation of Dennis Bernard Jeffrey is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I would like to sincerely thank my advisors, Dr. Neelam Gupta and Dr. Rajiv Gupta, for supporting me throughout this long journey. Six years ago, I knew nothing about graduate studies or research. My advisors were steadfast in providing me with invaluable help, guidance, and inspiration, and I never would have made it this far without them. They believed in me during times when I did not believe in myself. Most importantly, they deeply care about their students. I am immensely fortunate to be able to call them my advisors.

I would like to thank my other dissertation committee members, Dr. Gianfranco Ciardo and Dr. Iulian Neamtiu, for taking time out of their schedules to help me through this last stage of my journey. I know this dissertation will be all the better because of them.

I would also like to thank my lab-mates, particularly Vijay Nagarajan, Chen Tian, and Min Feng, for helping me in many ways during the last several years. They have been great friends and together we have shared many memories that I will never forget.

I would like to extend a general thank you to all of the other teachers I have had throughout my life. I am where I am today because of them.

Finally, I would like to thank my family. They are the ones who support me unconditionally. Always have, always will.

This dissertation is dedicated to my family:

past, present, and future.

ABSTRACT OF THE DISSERTATION

Dynamic State Alteration Techniques for Automatically Locating Software Errors

by

Dennis Bernard Jeffrey

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, August 2009
Dr. Rajiv Gupta, Chairperson

Software does not always behave as expected due to errors. These errors can potentially lead to disastrous consequences. Unfortunately, debugging software errors can be difficult and time-consuming. Many techniques to automatically locate errors have been developed, but the results are far from ideal. Unlike other techniques that analyze existing state information from program executions, dynamic state alteration techniques modify the state of program executions to gain deeper insight into the potential locations of errors. However, prior state alteration techniques are generally no more effective than other techniques, and come at the expense of increased computation time. This dissertation shows that *aggressive* and *well-targeted* state alteration techniques can be both highly effective and reasonably efficient.

The *Value Replacement* technique performs aggressive state alterations to locate software errors by replacing the set of values used in different statement instances in failing program executions. In a set of benchmarks, Value Replacement precisely identifies a faulty statement in 39 out of 129 cases, whereas the most effective technique previously known does so in 5 cases. Value Replacement can be generalized to iteratively locate multiple errors. A

brute-force implementation of Value Replacement can require hours to locate a single error, but techniques are developed that can reduce this timing requirement to minutes to locate multiple errors.

The *Execution Suppression* technique performs targeted state alterations to locate memory errors by iteratively suppressing (avoiding) the effects of statements involving known memory corruption during failing executions. The technique is able to precisely identify the first point of memory corruption in all analyzed benchmark programs; this point is typically at or close to the location of a memory error. Execution Suppression can be generalized to locate multithreading errors including data races. While a software-only implementation of suppression incurs an overhead of 7.2x on average, this overhead can be reduced to 1.8x using hardware support.

Finally, a machine learning technique called *BugFix* is developed that provides automated assistance in modifying a faulty statement to fix an error. A case study illustrates the potential benefit of the technique.

Table of Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**

1.1 The Problem of Software Errors 1

1.2 Handling Software Errors 3

1.2.1 Preventing Errors versus Responding to Errors 3

1.2.2 Overview of Software Testing and Debugging 5

1.2.3 Approaches for Automating Software Debugging 7

1.3 Dissertation Overview 11

2 Locating Errors using Value Replacement **17**

2.1 Computing Interesting Value Mapping Pairs 18

2.2 Examples of IVMPs Linked to Faulty Statements 22

2.2.1 IVMPs at a Faulty Statement 23

2.2.2 IVMPs Directly Linked to a Faulty Statement 23

2.2.3 IVMPs in the Presence of Erroneously-Omitted Statements 25

2.2.4 IVMPs in the Presence of Extraneous Statements 26

2.3 The Need to Consider Multiple Failing Runs 27

2.4 Ranking Statements using IVMPs 29

2.5 Effectiveness of Value Replacement 33

2.5.1 Setup for Experiments 33

2.5.2 Effectiveness Results and Discussion 39

2.5.3 Experiments with Larger Benchmark Programs 43

2.6 Summary 45

3 Value Replacement and Multiple Simultaneous Errors **48**

3.1 Techniques to Locate Multiple Errors 50

3.1.1 Minimal-Computation Technique 50

3.1.2 Full-Recomputation Technique 52

3.1.3 Partial-Recomputation Technique 53

3.2 Effectiveness Comparison of Techniques 59

3.2.1 Setup for Experiments . . . 59
3.2.2 Effectiveness Results and Discussion . . . 62
3.2.3 Comparison to a Clustering Technique . . . 63
3.3 Summary . . . 66

4 Efficiency of Value Replacement 67
4.1 The Cost of Value Replacement . . . 68
4.2 Lossy Techniques to Improve Efficiency . . . 70
4.2.1 Limiting the Number of Statement Instances to Consider . . . 71
4.2.2 Limiting the Number of Alternate Value Sets to Consider . . . 72
4.2.3 Value Replacement Algorithm with Lossy Efficiency Improvements . . 74
4.3 Lossless Techniques to Improve Efficiency . . . 77
4.3.1 Removing Redundant Program Execution . . . 78
4.3.2 Parallelizing the Search for IVMPs . . . 80
4.4 Efficiency Results . . . 81
4.4.1 Efficiency of Locating Single Errors . . . 81
4.4.2 Efficiency of Locating Multiple Errors . . . 85
4.5 Summary . . . 88

5 Locating Memory Errors using Execution Suppression 90
5.1 A Study of Memory Errors and Memory Corruption . . . 93
5.1.1 Memory Errors . . . 93
5.1.2 Results of the Study . . . 95
5.1.3 Key Observations . . . 98
5.2 Isolating the First Point of Memory Corruption using Suppression . . . 101
5.2.1 Motivational Example . . . 102
5.2.2 The Suppression Algorithm . . . 106
5.2.3 Real-world Example using the Suppression Algorithm . . . 109
5.3 Exposing Program Crashes using Variable Re-ordering . . . 112
5.3.1 The Variable Re-ordering Algorithm . . . 113
5.3.2 Example using the Variable Re-ordering Algorithm . . . 117
5.4 The Complete Execution Suppression Technique . . . 118
5.5 Evaluation of Execution Suppression . . . 119
5.5.1 Experiments with Suppression Only . . . 120
5.5.2 Experiments with Suppression and Variable Re-ordering . . . 121
5.6 Summary . . . 124

6 Execution Suppression and Multithreading Errors 127
6.1 Locating Multithreading Errors using Suppression . . . 129
6.1.1 Reproducing the Effects of Multithreading Errors . . . 132
6.1.2 On-the-fly Checking for Data Races . . . 134
6.1.3 Performing Suppression . . . 137
6.2 Illustrative Examples . . . 137
6.2.1 Example with Harmful Data Race Error . . . 137
6.2.2 Example with Non-Data-Race Error . . . 141
6.3 Evaluation using Multithreading Errors . . . 144

6.3.1 Setup for Experiments . . . 144
6.3.2 Results and Discussion . . . 146
6.4 Summary . . . 153

7 Suppression Implementation Issues 154
7.1 General Implementation . . . 155
7.2 Software-Only Implementation . . . 156
7.2.1 Basic Technique for Single-Threaded Programs . . . 156
7.2.2 Generalized Technique for Multithreaded Programs . . . 160
7.3 Hardware Support . . . 165
7.3.1 Using Existing Support for Deferred Exception Handling . . . 165
7.3.2 Additional Hardware Support through Memory Augmentation . . . 166
7.4 Overhead Comparison . . . 166
7.5 Summary . . . 168

8 Towards Correction of Program Errors 170
8.1 Association Rule Learning . . . 172
8.2 BugFix: Automated Assistance for Fixing Errors . . . 174
8.2.1 Analyzing the Debugging Situation . . . 175
8.2.2 Prioritizing Bug-Fix Suggestions . . . 183
8.2.3 Learning from the Debugging Scenario . . . 188
8.3 Case Study . . . 188
8.3.1 Training Phase . . . 189
8.3.2 Encountering New Debugging Situations . . . 192
8.4 Summary . . . 195

9 Related Work 197
9.1 Locating Errors . . . 197
9.1.1 Techniques for Locating Errors in General . . . 197
9.1.2 Techniques for Locating Multithreading Errors . . . 206
9.1.3 Techniques for Locating Specific Kinds of Errors . . . 208
9.2 Fixing Errors . . . 210
9.3 Tolerating Errors . . . 211
9.3.1 Avoiding the Negative Effects of Errors . . . 212
9.3.2 Recovering from the Negative Effects of Errors . . . 213

10 Conclusions 215
10.1 Contributions of this Dissertation . . . 215
10.2 Future Directions . . . 219

Bibliography 223

List of Figures

1.1 Focus of this dissertation in the context of handling software errors. 11
1.2 Organization of this dissertation. 16

2.1 Example IVMPs at a statement. 20
2.2 General algorithm for computing IVMPs in a failing run. 22
2.3 Code fragment based on `schedule`, faulty version v9. 24
2.4 Code fragment based on `tcas`, faulty version v7. 25
2.5 Code fragment inspired by `schedule2`, faulty version v1. 26
2.6 Example to motivate the need to consider multiple failing runs. 29
2.7 The Value Replacement technique. 32
2.8 Comparison of statement ranking techniques. 41
2.9 Increase in value profile size as suite sizes increase. 44

3.1 Single-fault core technique and multi-fault generalized techniques. 51
3.2 Minimal-Computation technique to locate multiple errors. 52
3.3 Full-Recomputation Technique to locate multiple errors. 53
3.4 Partial-Recomputation technique to locate multiple errors. 54
3.5 Example for Partial-Recomputation technique, part 1 of 2. 57
3.6 Example for Partial-Recomputation technique, part 2 of 2. 58

4.1 Total number of program executions for full IVMP search. 70
4.2 The Value Replacement technique with lossy efficiency improvements. 75
4.3 Lossless improvements to the efficiency of Value Replacement. 79
4.4 Total number of program executions for full vs. reduced IVMP search. 82
4.5 Time required to search for IVMPs using the reduced search. 84
4.6 Average total time to search for IVMPs. 86

5.1 Example to illustrate suppression. 103
5.2 Suppression executions for the example program. 104
5.3 The suppression algorithm. 107
5.4 A failing execution in the `pine` program. 110
5.5 General variable re-ordering algorithm. 114
5.6 Example to illustrate variable re-ordering. 118
5.7 The complete Execution Suppression algorithm. 119

6.1 Extended Execution Suppression algorithm for multithreading errors. 131
6.2 On-the-fly checking for data races. 136
6.3 Example multithreaded code snippet, part 1. 138
6.4 Example executions for running technique with a harmful data race error. . . 139
6.5 Example multithreaded code snippet, part 2. 142
6.6 Example executions for running technique with a non-data-race error. 143

7.1 General implementation of suppression. 156
7.2 High-level design for the basic technique software implementation. 157
7.3 High-level design for the generalized technique software implementation. 161
7.4 Itanium processor with and without memory augmentation. 167
7.5 Execution time overheads for different suppression implementations. 168

8.1 The three main steps of BugFix. 175
8.2 Deriving situation descriptors from `C` program structure. 178
8.3 Example of `C` structure situation descriptors. 179
8.4 Example of identifying situation descriptors in IVMPs. 181
8.5 Example of identifying situation descriptors in exercised value sets. 183
8.6 Example of three rules from a knowledgebase of rules. 186
8.7 Example of prioritizing bug-fix suggestions for a debugging situation. 187
8.8 Four faulty program debugging scenarios used to train BugFix. 191
8.9 Four new debugging scenarios (post-training) for the case study. 193

List of Tables

2.1 The Siemens benchmark programs. 35
2.2 Number/score of ranked statement lists for basic techniques. 40
2.3 Number/score of ranked statement lists for variation techniques. 40
2.4 Larger benchmark programs. 44
2.5 Experimental results using the larger benchmark programs. 45

3.1 Multiple-error experimental subjects. 61
3.2 Average score achieved for each located error using each technique. 63
3.3 Average score for each located error using clustering. 65

4.1 Efficiency results for larger benchmarks using the reduced search. 84
4.2 Average number of runs searched (separated by iteration number). 88

5.1 Memory error programs analyzed in the memory corruption study. 96
5.2 Study results for each memory error subject program. 97
5.3 Experimental results using only suppression. 121
5.4 Experimental results using suppression and variable re-ordering. 122

6.1 Benchmark programs used in the experiments. 145
6.2 Results of running the technique to locate multithreading errors. 146

8.1 Siemens benchmark programs used in the case study. 189
8.2 Bug-fix descriptions involved in this case study. 192

Chapter 1

Introduction

1.1 The Problem of Software Errors

According to the *New York Times* [61], the term "software" was coined by a statistician named John Wilder Tukey in a 1958 article in *American Mathematical Monthly*. This article was published just 51 years ago. Since then, the use of computers and computer software has steadily become more and more prevalent in society. The last few decades in particular have seen a dramatic increase in the amount of software in use around the world, to a level that likely could not have been envisioned in 1958. From education to the economy, the military to medicine, and the arts to entertainment, software has become a tool upon which society relies in order to function. The many benefits of software are too numerous to list. However, with these benefits come important challenges that must be addressed. One of the most important of these challenges is the fact that software does not always behave as is intended. In other words, software does not always behave in a reliable manner. Given the ubiquity of computer software in use in today's society, the consequences of unreliable

software can be disastrous. These consequences can be measured in terms of both the financial cost as well as the cost in human life.

On June 4, 1996, the first test flight of the European *Ariane 5* expendable launch system was conducted. Less than a minute after launch, the rocket deviated from its intended flight path and was destroyed, resulting in a loss of more than $370 million [26]. One of the software problems leading to this disaster was an arithmetic overflow during a data conversion from a 64-bit floating point to a 16-bit signed integer. In September of 1999, the $125 million *Mars Climate Orbiter* was lost upon entering the atmosphere of Mars. The cause was determined to be a navigation error caused by the use of Imperial units rather than the metric system [54]. On February 25, 1991, a U.S. *Patriot* missile failed to intercept an incoming Iraqi *Scud* missile, which exploded at an Army base in Dhahran, Saudi Arabia, killing 28 soldiers and wounding about 100 others [154]. The cause was determined to be a rounding error in software, resulting in a clock skew of 0.34 seconds that caused a miscalculation of the incoming missile's location. Between June of 1985 and January of 1987, the *Therac-25* radiation therapy machine administered deadly radiation overdoses to several cancer patients, estimated to be more than 100 times greater than the dosage typically used for treatment [154]. The cause of this tragedy was determined to be a software race condition.

One of the primary causes of software unreliability is the presence of errors in program source code. Unfortunately, errors occur frequently in software. It is common practice for businesses to release software that contains known errors, simply because there is often not enough time or resources to address all errors. In these cases, errors are usually prioritized so that high-priority errors can be fixed prior to release. Even if all known errors

can be fixed prior to release, some errors may not reveal themselves until after software has been deployed and is in use – sometimes to disastrous effect.

The main reason why software errors are commonplace is that programming is a human-intensive and very complicated activity. It is often very difficult or seemingly impossible to reason about all possible executions of a program, and to foresee all possible environmental factors that may influence, or be influenced by, a program. Even when a program appears to behave properly when executed, there may still exist subtle errors that may only appear on rare occasions or when a certain set of conditions are met. According to a June 2002 report by the *National Institute of Standards and Technology* (NIST), software errors "are so prevalent and so detrimental that they cost the U.S. economy an estimated $59.5 billion annually," with over half of these costs being borne by software users [60]. As the amount of software in use around the world increases, and the degree to which humanity relies on software rises, the problem of software errors is becoming all the more important. Thus, the reality of software errors is an urgent problem with far-reaching and potentially-disastrous consequences.

1.2 Handling Software Errors

1.2.1 Preventing Errors versus Responding to Errors

There are two general approaches that can be used to address the problem of software errors. The first approach is to develop techniques that can help prevent errors from happening in the first place. The second approach is to deal with the errors after the fact: to discover and eliminate the errors, or to otherwise avoid or tolerate their effects.

Preventing errors involves modifying the software development process to decrease the chances that errors will be introduced into software. For example, *rigorous software engineering* [12] emphasizes the early stages of the software development process, advocating for an early, precise understanding of the required behavior of the software so as to "get it right the first time." Another idea, *automated error prevention* [57], is a framework that correlates errors found in the development process with the development procedures that were responsible for creating them; these development procedures are then modified to prevent the error from recurring in the future. To borrow an analogy from Adam Kolawa, co-founder and CEO of Parasoft Corporation, the process of error prevention is beneficial because it treats the disease (the development process), rather than merely the symptom (the errors themselves) [57].

Developing techniques to prevent the creation of errors can greatly improve the reliability of developed software. However, many would agree that this process – at least in the near future – will never completely eradicate errors introduced during software development, simply because software is still developed by humans, and humans are prone to error. Moreover, preventing errors in the development process does not address the fact that many errors already exist in developed and deployed software. Thus, in the foreseeable future, there will always be a need to deal with existing errors. One way to deal with errors is to tolerate them, or to otherwise avoid the negative effects of the errors during execution, without actually eliminating the errors themselves. Another option, which is the best option for promoting program reliability, is to identify and eliminate the errors. To accomplish this, the approaches of *software testing* and *software debugging* often go hand in hand.

1.2.2 Overview of Software Testing and Debugging

Software testing refers to the process of analyzing software to obtain information about its potential correctness. It may be performed statically, through techniques such as code inspection, code reviews, and code walkthroughs. However, it is more commonly performed dynamically, by executing software using different input values, and then examining the behavior of the software to determine if the actual behavior matches the expected behavior. Typically, this involves checking whether the output of the software matches the output that is expected. Testing is performed in order to try to expose software errors and gain confidence in the correctness of a program. The more software behaves as intended during the testing process, the more confidence one can have in its correctness. Whenever any unexpected behavior arises, then a developer can determine if the unexpected behavior is due to an error in the software; if so, then the error can be fixed and testing can resume. In general, extensive testing may provide high confidence that a program is correct. However, testing cannot guarantee program correctness because it is usually infeasible to execute a program under all possible input values and under all possible conditions. Nevertheless, testing is an important phase of software development for improving the reliability of software.

While software testing can be used to expose errors in software, *software debugging* is the process of actually removing those errors from the software (i.e., fixing the errors). Debugging can be a challenging task because it is composed of two general steps, either of which can be very difficult in certain situations.

Once an error is known to exist in software, then the error must first be located in the software. In some cases, this step can be very time consuming because the point at

which an error is revealed during program execution may be far away from the point of the error itself. For example, an error may be traversed early on during program execution, and this may eventually result in incorrect output being produced at a point much later during execution. Once a developer observes the incorrect output, it may take considerable time to isolate the error in the source code that originally caused the incorrect output.

After an error is located, the nature of the error must be fully understood so that a developer can modify the source code to eliminate the error. In some cases, the nature of the error may be obvious and the appropriate fix may be determined quickly. However, in other cases, the appropriate fix to make may not be so clear. Even if an attempt is made to fix an error, it might turn out that the attempted fix leads to unintended consequences that may cause the software to misbehave in other ways. In other words, the process of debugging can sometimes introduce additional errors. Because of this, debugging can be a very difficult and time-consuming task.

Software testing and debugging are important and necessary phases of software development, and they work together to promote robust and reliable software: testing is used to expose software errors, and debugging is used to eliminate those errors. Because both of these processes are important, there is significant prior research work in both areas. However, in one sense, debugging currently poses a more pressing challenge because the rate at which software errors are eliminated through debugging often cannot keep up with the rate at which software errors are discovered through testing. Evidence of this can be seen by companies releasing software with known errors, often with the promise of releasing *patches* for the software in the future once the errors can be properly addressed.

The main reason why software debugging is so time-consuming is that it usually

involves significant manual work. In most cases, developers must manually locate errors, understand them, and determine how to modify source code to eliminate them. Because of this, there has been significant work that focuses on automating the debugging process. Providing automated assistance for debugging can improve the efficiency of the debugging process, allowing more errors to be fixed in shorter time, thus promoting more robust and reliable software.

1.2.3 Approaches for Automating Software Debugging

There have been a few techniques developed to automatically assist developers in fixing software errors. He and Gupta [46, 47] developed an approach that uses the notion of path-based weakest preconditions to automatically generate program modifications to correct an erroneous statement in a function. The function must be associated with a formal precondition and postcondition. Abraham and Erwig developed a debugging tool that can assist in correcting errors in spreadsheets [1]. In this tool, a user specifies the expected value for a cell that contains an incorrect value; the tool then identifies change suggestions that can be used to correct the error.

These techniques provide automated assistance for developers to modify software to eliminate errors. However, the amount of research in this area is relatively little as compared to the research on automatically locating software errors. One of the main reasons why there is little prior work on automatically assisting in fixing errors is because this is a difficult task to automate. In general, automatically fixing an error would require a tool to know the intended behavior of software, which would have to be formally specified in some way; this can place burden on a developer to formally provide this information. Moreover,

fixing an error also requires knowledge about an appropriate location in the source code at which a fix can be made. This can be the location of an error itself, or else another location at which a source code modification can offset the negative effects of an error. In other words, error location is, in general, a prerequisite to error correction. As a result, most prior work on automating the debugging process has focused on the task of locating faulty program statements.

One thread of research involves *slicing-based* approaches that can be used to narrow down the search for faulty program statements. *Static Slicing* [139] identifies a subset of program statements that may influence the value of a variable at a program location, obtained by static analysis of a program. *Dynamic Slicing* [3, 78, 133, 149, 150] identifies a subset of program statements that actually do influence the value of a variable at a particular point in a given dynamic program execution. The related concept of *Relevant Slicing* [4, 40] has also been studied to incorporate potential dependencies. In general, slices computed from the point of an incorrect program value can identify the subset of statements that could have contributed to that incorrect value; this subset of statements is likely to contain a faulty statement, and may be significantly smaller than the set of all program statements.

Another thread of research on automatically locating program errors involves *statistical analysis*. Some of these approaches [74, 75, 87, 88] use dynamic information obtained from program executions to rank program statements according to likelihood of being faulty. Jiang and Su [72] proposed a context-aware approach that constructs faulty control flow paths linking bug predictors together, that can be used to help explain software errors. The *Nearest Neighbor* technique [119] searches for a passing execution that is most similar to a failing execution, compares the spectra for these two executions, and uses this information

to identify the most suspicious parts of the program.

Yet another thread of research on error location involves the concept of *state alteration*. These techniques modify the state of an executing program in an attempt to isolate software errors. In the *Delta Debugging* framework, failure-inducing input is identified [146] that allows for the computation of cause-effect chains for failures [145] that can in turn be linked to faulty code [22]. This is accomplished by swapping program state (the values of variables) between a successful and failing run. *Predicate Switching* [148] attempts to isolate erroneous code by identifying predicates whose outcomes can be altered during a failing run to cause the run to become passing.

State alteration techniques are particularly promising for effectively locating software errors. This is because other techniques (that do not alter state) only consider static information and/or the dynamic information from a set of program executions that are associated with available test cases. These techniques are limited because they consider only *existing* static and dynamic information that is already available. State alteration techniques, on the other hand, can also consider *new* dynamic information that can be observed by *modifying* the state of existing program executions. Each time the state of an executing program is altered, this leads to an entirely new execution that can be analyzed. In many cases, these state alterations can provide important insights about software errors that can assist in debugging them.

Although there is significant prior work on automated assistance for locating software errors, the problem of error location is by no means a solved problem. Ideally, an automated technique should precisely report the location of any software error. However, in practice, the ideal situation rarely occurs. Current techniques report a subset of program

statements (that may be priority-ordered) showing the most suspicious parts of a program, and a developer must usually examine multiple statements until an error is found. There is much room for improvement in the results reported by these techniques.

This dissertation supports the notion that there is great potential in dynamic state alteration techniques for automatically locating software errors. However, current state alteration techniques have not fully explored the potential of state alteration, and so the effectiveness of current techniques in locating errors is not nearly as high as what may be possible with state alteration. For example, both Delta Debugging and Predicate Switching involve different types of state alteration: Delta Debugging alters data-flow state while Predicate Switching alters control-flow state. However, Delta Debugging was shown [74] to be less effective at locating errors than a much simpler statistical technique for a particular set of benchmark programs. Moreover, Predicate Switching can only identify "critical" predicates whose outcomes can be altered to correct the output of a failing run; the technique will likely be ineffective for errors that do not influence control flow. Even if a critical predicate is found, the actual error may be elsewhere in a non-predicate statement.

The effectiveness of existing state alteration techniques in locating errors may be limited due to a few reasons. First, the techniques may not be *aggressive* enough in their state alterations to gather the most relevant and useful information about the likely locations of program errors. Second, the state alterations performed by existing techniques may not be ideally *targeted* to achieve the desired error location results. This dissertation considers how more aggressive and better-targeted state alteration can improve the effectiveness of locating software errors. Figure 1.1 illustrates the focus of this dissertation in the context of handling software errors.

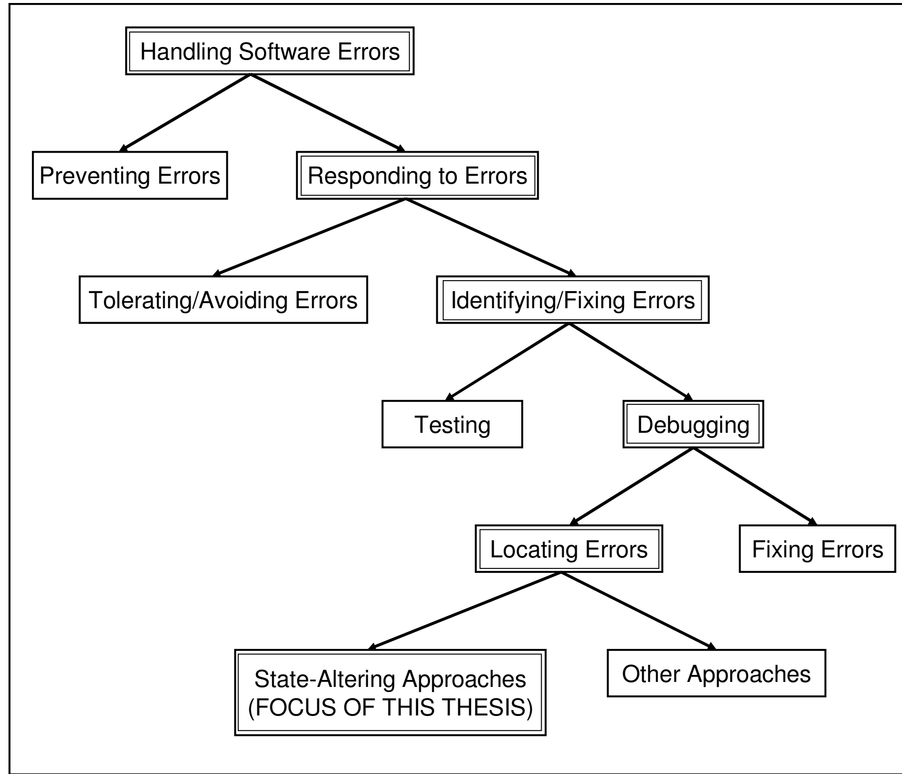

Figure 1.1: Focus of this dissertation in the context of handling software errors.

1.3 Dissertation Overview

The goal of this dissertation is to show that dynamic state alteration techniques for locating software errors can be both highly effective and efficient enough for practical use in a debugging context. To this end, this dissertation develops two new dynamic state alteration techniques for locating errors.

The first state alteration technique developed in this dissertation is called *Value Replacement*. This technique ranks program statements according to how likely they are to be faulty, thus assisting a developer in quickly locating a software error. Value Replacement aggressively alters the execution state at different statement instances in a failing program execution, by replacing the set of values involved in a statement instance with an alternate

set of values. After each value replacement at a statement instance, the output of the execution is examined to determine whether it has changed to become correct. If so, then the statement associated with the value replacement is likely to be associated with an error. This technique performs aggressive state alteration because it considers multiple statement instances in multiple failing runs, and performs value replacements with different alternate value sets at each statement instance. A generalized version of Value Replacement is described that can handle the case when multiple errors exist simultaneously in software. Since a brute-force implementation of Value Replacement can be very time-consuming, techniques are also developed for significantly improving the efficiency of Value Replacement.

The second state alteration technique developed in this dissertation is called *Execution Suppression*. This technique involves two state alteration ideas that work together to isolate memory errors: *suppression* and *variable re-ordering*. Suppression identifies known memory corruption that is revealed by a program crash, and omits (suppresses) the direct and indirect effects of the associated statements during execution. This bypasses the initial crash and gives any remaining memory corruption an opportunity to cause subsequent program crashes. This process iterates until the first point of memory corruption is revealed, which is assumed to be at or near to a memory error. The state alteration performed by suppression is targeted at those statement instances that are directly or indirectly influenced by known memory corruption during execution. Variable re-ordering is used to expose crashes due to memory corruption in cases where crashes do not otherwise occur. This is achieved by altering the relative ordering of particular variables in memory prior to execution. An extended version of Execution Suppression is developed that can handle memory errors in multithreaded programs, including data race errors. Finally, implementation issues for

suppression, including hardware support, are considered.

Since locating a software error is only the first step in fixing the error, this dissertation develops a technique called *BugFix* that can automatically assist developers in fixing program errors. The technique identifies and reports a prioritized list of suggestions for how to modify a particular statement to fix an error in that statement. The technique is based upon a machine-learning algorithm that incorporates information obtained from a history of prior faulty statements and their associated fixes. This allows the technique to become more effective over time. BugFix is inspired by the work done on Value Replacement.

The contributions of this dissertation are summarized as follows.

Value Replacement

- A new state alteration technique called *Value Replacement* is developed that can be used to locate software errors. The technique replaces the set of values involved at different statement instances in failing executions with alternate sets of values. Any such replacements that cause the output of an execution to change to become correct, are likely to be associated with a software error.

- Three generalized versions of Value Replacement are developed that can be used to effectively locate multiple errors when they exist simultaneously in software. The techniques iteratively present a ranked list of program statements to a developer to find and fix one error at a time.

- The effectiveness of Value Replacement is evaluated in an empirical study. It is shown that Value Replacement can achieve significantly better error location results on a set of benchmark programs than a prior statistical technique called *Tarantula* [75]. Taran-

tula had previously been shown [74] to be more effective than several other existing techniques (including a state-altering technique [22]) on the same set of benchmark programs.

- Several techniques are presented to improve the efficiency of Value Replacement. Some of these techniques improve efficiency without any loss in effectiveness. Other techniques may result in slightly diminished effectiveness, but can drastically improve the efficiency of the technique. Experimental results regarding the efficiency of Value Replacement are presented.

Execution Suppression

- A new state alteration technique called *Execution Suppression* is developed that can be used to locate memory errors in software. The technique uses the notion of suppression to omit (suppress) the effects of a known memory corruption during execution to iteratively reveal more memory corruption, until the first point of memory corruption can be identified. Memory corruption is revealed during execution by a program crash. The technique uses the notion of variable re-ordering to expose program crashes due to memory corruption, in cases where crashes may not otherwise occur.
- An extended version of Execution Suppression is developed that can be used to effectively locate memory errors – including data race errors – in multithreaded programs. The technique checks whether a data race is potentially harmful before reporting it as the likely error that caused a program failure.
- The effectiveness of Execution Suppression is evaluated in an empirical study. It is

shown that the technique is highly effective at precisely identifying the first point of memory corruption in executions that fail due to memory errors, and that these located points are always either at, or close to, the memory errors themselves.

- Implementation issues regarding suppression are described. A software-only implementation is discussed, and certain kinds of hardware support are also considered. Experimental results comparing the overheads of these different implementations are provided.

BugFix

- A machine learning technique called *BugFix* is developed that can be used to automatically assist developers in fixing program errors. The technique identifies and reports a prioritized list of suggestions for how to modify a given suspicious statement in order to correct a potential error in that statement.

- A detailed case study is described that illustrates the potential benefit of BugFix.

The rest of this dissertation is organized as depicted in Figure 1.2. The Value Replacement state alteration technique is developed in the next chapter, and an empirical evaluation of the effectiveness of the technique is presented. In Chapter 3, generalized versions of Value Replacement for handling multiple simultaneous errors are developed, and the effectiveness of the different versions are compared. In Chapter 4, efficiency issues related to Value Replacement are described; techniques for improving efficiency are developed, and experimental results regarding the efficiency of the technique are presented. The Execution Suppression state alteration technique is developed in Chapter 5, and an empirical study

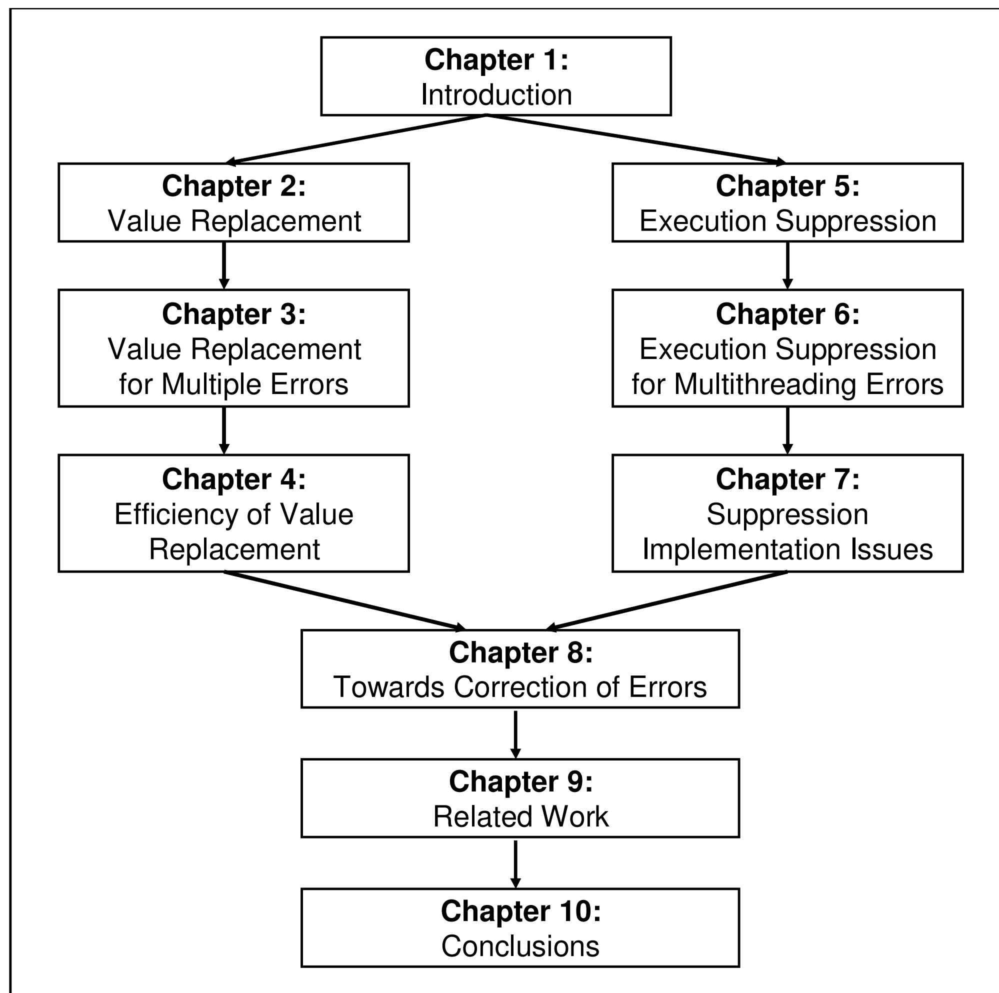

Figure 1.2: Organization of this dissertation.

that evaluates the technique is presented. In Chapter 6, an extended version of Execution Suppression is developed that can handle multithreading errors such as data races, and an empirical evaluation is presented. In Chapter 7, software and hardware implementation issues for suppression are described, and the overheads associated with several different implementations are compared. The BugFix technique for providing automated assistance in fixing faulty program statements is developed in Chapter 8. Related work is presented in Chapter 9. The conclusions of the dissertation are summarized in Chapter 10.

Chapter 2

Locating Errors using Value Replacement

In this chapter, an automated, dynamic state alteration technique called *Value Replacement* is developed for locating software errors. This technique analyzes program executions that fail due to incorrect output being produced. In such a failing execution, Value Replacement alters the execution state at a single statement instance, one after the other, by replacing the set of values involved at that statement instance with an alternate (different) set of values. Execution then proceeds from that point under the altered state. At the end of execution, the output is examined to determine whether or not it has changed to become correct. If the output has become correct, then there is a chance that the statement instance at which the value replacement was performed, is faulty. The Value Replacement technique performs these value replacements at different statement instances in a failing execution, one at a time, to rank program statements according to how likely they are to be faulty. This is the essence of the Value Replacement technique.

Definition 1 (Value Replacement).

Given a statement instance in the execution of a failing run, a **value replacement** *involves replacing the set of values involved at that statement instance with an alternate (different) set of values. Execution then proceeds from that point under the altered state until the execution terminates. Program state that is altered under a value replacement can consist of global, local (stack), and heap values. Address values are not considered when performing value replacements.*

2.1 Computing Interesting Value Mapping Pairs

If a value replacement causes the execution of a failing run to become correct, this fact is represented by an *interesting value mapping pair* (IVMP). An IVMP is associated with a statement instance in a failing execution, and is composed of two sets of values: the original set of values used at that statement instance, and the alternate set of values that can be substituted in place of the original values in order to cause the execution to produce correct output. An IVMP is a "value mapping pair" because it is composed of a pair of value mappings (sets of values). An IVMP is "interesting" because it represents how the state of a failing execution can be modified in order to cause the execution to become passing (i.e., produce correct output).

Definition 2 (Interesting Value Mapping Pair).

An **interesting value mapping pair (IVMP)** *is a pair of value mappings ("original", "alternate") associated with a particular statement instance in a failing run, such that: (1) "original" is the original set of values used by the failing run at that instance; and (2)*

"alternate" is an alternate (different) set of values such that if the values in "original" are replaced by the values in "alternate" at that instance during execution of the failing run, then the incorrect output of the failing run becomes correct.

To illustrate, Figure 2.1 shows three possible IVMPs for a given statement instance. The statement in this case is an `if` condition in which the $<$ operator is mistakenly used instead of $<=$. The effect of this error is that whenever the operand values x and y are identical, then the condition will erroneously evaluate to `false` when it should have evaluated to `true`. As a result, all original sets of values in the IVMPs have identical values for x and y, and the condition evaluating to `false`. However, all alternate sets of values have different values for x and y that instead cause the condition to evaluate to the expected outcome of `true`. These alternate values can cause a failing run to pass (assuming, for instance, that neither x nor y are subsequently referenced and that there are no other errors in the program).

The Value Replacement technique involves searching for IVMPs that can be associated with a failing run. If a program statement is associated with at least one IVMP, then this statement can be shown to affect the output of a failing run such that the incorrect output becomes correct. The intuition is that these statements are more likely to be faulty, as compared to other statements that are not associated with any IVMPs. As will be discussed in detail later, this IVMP information is used to rank program statements according to how likely they are to be faulty.

Given a failing run, the task of searching for IVMPs is straightforward: simply consider statement instances in the failing run one at a time, replacing the value mapping used at each one with a different value mapping, then checking to see if the output of the

Suspicious Statement:

```
// assume that '<' should actually be '<='
if (x < y)
```

Three Possible IVMPs:

```
(1) ORIGINAL:  {x=1, y=1, branch=FALSE}
    ALTERNATE: {x=3, y=5, branch=TRUE}

(2) ORIGINAL:  {x=8, y=8, branch=FALSE}
    ALTERNATE: {x=1, y=2, branch=TRUE}

(3) ORIGINAL:  {x=3, y=3, branch=FALSE}
    ALTERNATE: {x=12, y=82, branch=TRUE}
```

Figure 2.1: Example IVMPs at a statement.

run becomes corrected. If so, an IVMP has been found. Searching for IVMPs requires only a failing test case execution with the corresponding incorrect and correct outputs, and some set of alternate value mappings that can be applied at different statement instances in the failing execution.

In general, the set of all possible alternate value mappings at a statement instance can be theoretically infinite. For example, suppose the value 1 is used at a statement instance. Then the set of all possible alternate values is the set of "all values except 1," which is, in theory, an infinite set (in practice, the size of the set would be limited by the maximum number of values that can be stored in the associated storage location). It is impractical to perform a value replacement for every possible set of alternate values. A method is required to select a finite set of alternate mappings that can be applied at each statement. This is accomplished by extracting the (finite) set of alternate mappings for each statement from the execution traces of all test cases in an available test suite. This includes the test case associated with the failing execution being analyzed, since different instances of the current statement in the same execution, may involve different values. The

extracted set of alternate mappings is called the *value profile*.

Definition 3 (Value Profile).

A **value profile** *for a program with respect to a test suite is a mapping of each program statement to the set of all unique sets of values occurring at that statement during execution of test cases in the test suite.*

It is reasonable to assume the existence of a test suite for computing the value profile since a failing test case is usually part of a larger suite of test cases. It has been observed [67] that rich value profiles can result from only a few test cases, and yet the sizes of value profiles increase logarithmically in general as the number of test cases in the suites increase. This is because as information from more test cases is added to a value profile, the sets of values used by a test case tend to match those already added to the value profile from previous test cases. In the value profile, the alternate sets of values between passing and failing executions are not distinguished. This is because alternate sets of values that may result in IVMPs can potentially come from *any* test case executions, regardless of whether the executions pass or fail.

The general algorithm for searching for IVMPs is given in Figure 2.2. Given a test suite with a failing test case for some program, the first step constructs the value profile using the traces of the test cases in the test suite. The second step searches for IVMPs by replacing the value mapping at each statement instance in the failing execution with every alternate value mapping from that statement as specified in the value profile. The runtime of this algorithm is therefore bounded by $O(t \times m)$, where t is the number of statement instances in the execution trace of the failing run, and m is the maximum number of alternate mappings for any statement in the value profile. Although this algorithm shows

```
input:
      Faulty program P, and failing test case f (with actual and expected output)
      from test suite T.
output:
      Set of identified IVMPs for f.
algorithm SearchForIVMPs
begin
Step 1: [Compute value profile for P with respect to T]
1:      valProf := {};
2:      for each test case t in T do
3:         trace := trace of value mappings from execution of t;
4:         augment valProf using the data in trace;
        end for
Step 2: [Search for IVMPs in f]
5:      trace_f := trace of value mappings from execution of f;
6:      for each statement instance i in trace_f do
7:         origMap := value mapping from trace_f at i;
8:         s := the statement associated with instance i;
9:         for each altMap in valProf at s do
10:           execute f while replacing origMap with altMap at i;
11:           if output of f becomes correct then
12:              output IVMP (origMap, altMap) at i;
           end for
        end for
end SearchForIVMPs
```

Figure 2.2: General algorithm for computing IVMPs in a failing run.

how to compute all possible IVMPs for a failing run (with respect to a particular test suite), a method to more efficiently search for a much smaller subset of IVMPs that are still effective for locating errors is developed in Chapter 4.

2.2 Examples of IVMPs Linked to Faulty Statements

It has been seen [67] that IVMPs occur precisely at faulty statements in many cases. In cases where this is not possible, IVMPs can occur at statements that are just one static dependence edge away from faulty statements. Because of this, IVMPs can be useful

for locating errors. Several examples are now presented that show different ways in which IVMPs can be closely linked to faulty statements. These examples are based on situations that are encountered using the Siemens benchmark programs [64].

2.2.1 IVMPs at a Faulty Statement

IVMPs can be found precisely at a faulty statement when applying an alternate set of values causes the faulty statement to define the correct value. Figure 2.3 shows a code fragment and a test suite based on Siemens program `schedule`, faulty version v9. This fragment of code involves a check on the number of input arguments (*argc*), so that the program terminates with an error message if there are too few input arguments specified. There is an off-by-1 error in this condition.

The effect of this off-by-1 error is that when *argc* is equal to 3, the program will erroneously proceed as normal when it should have terminated early due to too few input arguments. Thus, executing test case B in Figure 2.3 results in a failure. However, for test case B, changing the value of *argc* at line 1 from 3 to 2 (which is the value used by test case A) causes the output of the failing run to become correct. Therefore, this represents an IVMP providing an important clue that at line 1 in the code fragment, the value of variable *argc* should be decremented by 1 (or equivalently, the value of constant 3 should be incremented by 1). In this case, the IVMP is located at precisely the faulty statement.

2.2.2 IVMPs Directly Linked to a Faulty Statement

In some cases, IVMPs may not be found precisely at the faulty statements. One situation where this can happen is when there is an error in a constant assignment statement.

```
    argc := ...;
1:  if (argc < 3)      /* 3 should actually be 4 */
2:     print ("Too few");
3:  else
4:     print ("Okay");
```

Test Case	Input Values	Actual Output	Expected Output	Result
A	$argc = 2$	Too few	Too few	PASS
B	$argc = 3$	Okay	Too few	FAIL
C	$argc = 4$	Okay	Okay	PASS

Figure 2.3: Code fragment based on `schedule`, faulty version v9.

A constant assignment will never be associated with an IVMP because there are no alternate values at the assignment; every executed instance of a constant assignment will define the same constant value. Instead, IVMPs can be found at the statements in which the defined constant values are used. Figure 2.4 shows a code fragment and test suite based on Siemens program `tcas`, faulty version v7. The code fragment shows an erroneously-defined constant value at line 2, which is larger than it should be.

The effect is that when the array index $AltLayVal$ is 1, the condition at line 5 will erroneously evaluate to `false` instead of `true` due to the incorrect constant value defined at that position. Thus, executing test case B in Figure 2.4 results in a failing run. However, for test case B, changing the value of $AltLayVal$ at line 5 from 1 to 0 (which is the value used by test case A) causes the output of the failing run to become correct. Assuming the value of index variable $AltLayVal$ is correct, this IVMP provides the important clue that the value stored at array index 1 is incorrect. Further, since accessing array index 0 (with value 400) corrects the output of the failing run, this provides the hint that the value 550 at array index 1 should be changed to another value that is smaller. In this case, the IVMP

```
    AltLayVal := ...;
1:  Pos_RA_Alt_Thresh[0] = 400;
2:  Pos_RA_Alt_Thresh[1] = 550; /* Should be 500 */
3:  Pos_RA_Alt_Thresh[2] = 640;
4:  Pos_RA_Alt_Thresh[3] = 740;
    ...
5:  if (Pos_RA_Alt_Thresh[AltLayVal] < 525)
6:    print (0);
7:  else
8:    print (1);
```

Test Case	Input Values	Actual Output	Expected Output	Result
A	$AltLayVal = 0$	0	0	PASS
B	$AltLayVal = 1$	1	0	FAIL
C	$AltLayVal = 2$	1	1	PASS

Figure 2.4: Code fragment based on `tcas`, faulty version v7.

is located at line 5, which is one data dependence edge away from the faulty statement at line 2.

2.2.3 IVMPs in the Presence of Erroneously-Omitted Statements

Another situation in which IVMPs cannot be found at an erroneous statement is when the error involves one or more missing statements. In these cases, IVMPs can still be found at nearby statements that can compensate for the effects of the missing code. Figure 2.5 shows an erroneous function and accompanying test suite inspired by `schedule2`, faulty version v1. The purpose of this function is to return the inputted value of x incremented by one, only when the value of y is positive (in bounds). If y is equal to 0, the function returns 0.

The missing code at line 1 is meant to check whether y is negative (out of bounds), and if so, to return the original value of x without having incremented it. Since test case A

```
    int foo(int x, int y)
1:    /* if (y < 0) return x; */
2:    if (y == 0) return 0;
3:    return x + 1;
```

Test Case	Input Values	Actual Output	Expected Output	Result
A	$(x,y) = (1,-1)$	2	1	FAIL
B	$(x,y) = (2,2)$	3	3	PASS
C	$(x,y) = (0,1)$	1	1	PASS

Figure 2.5: Code fragment inspired by `schedule2`, faulty version v1.

has y with out-of-bounds value -1, then the function erroneously increments the value of x in this case when it should not have done so. When the value of x at line 3 is changed from 1 to 0 (which is the value used by test case C), then the output becomes 1 and is correct. This IVMP at line 3 provides the important clue that for the failing run corresponding to test case A, the value for x actually should not have been incremented. This suggests that a statement (the one at line 1) is missing in the above function that will prevent test case A from incrementing the value of x.

2.2.4 IVMPs in the Presence of Extraneous Statements

Some program errors may involve extraneous statements. It turns out that IVMPs often occur precisely at extraneous statements where they have the effect of "canceling out" the effects of the extra code. For instance, an extraneous assignment statement to variable x can have an IVMP that forces the original value of x to be defined, rather than the new value for x that resulted from the extra code. An extraneous condition can have an IVMP that alters the conditional outcome in such a way that the behavior is as if the condition is not present.

2.3 The Need to Consider Multiple Failing Runs

Although IVMPs often occur at or near faulty statements, a significant challenge to using IVMPs for locating errors is that IVMPs can be found at other statements besides those that are faulty. This is possible because there are often multiple statements exercised during a failing execution whose values can be changed to cause the output to become correct. There are two main causes for this, referred to as the *dependence cause* and the *compensation cause* for IVMPs at multiple statements.

Dependence Cause. IVMPs may be found at different statements that are all part of the same definition-use chain in a program. This is because if a statement S_1 defining a variable x has an IVMP associated with it, there's a chance that another statement S_2 that uses x will also have an IVMP associated with it. In such cases, changing the value of x at either S_1 or S_2 can correct the program output, even though only one of the two statements may contain an error.

Compensation Cause. This occurs when IVMPs are found at two different statements that do not appear to be related to each other at all, yet they both influence the output such that applying an alternate set of values at either statement can compensate for the effects of the error on the program output, thereby making the output correct.

To address the challenge posed by the dependence and compensation causes for IVMPs at multiple statements, the technique considers IVMPs computed from *multiple* failing runs. A dependence chain with IVMPs in one failing run may not exist in another failing run that may involve different dependence chains. Also, IVMPs that happen to compensate for an error in one failing run are unlikely to compensate for the error in the same way in another failing run. Considering multiple failing runs is particularly effective

when the failing runs exercise very different paths in the program. Since all failing runs must traverse the error (assuming a single error exists), the statements that are associated with IVMPs in more failing runs have a greater likelihood of being faulty. Therefore, IVMP statements are ranked using the intuition that statements associated with IVMPs in more failing runs are more likely to be faulty, than statements that are associated with IVMPs in fewer failing runs. Consider the example program with accompanying test suite in Figure 2.6.

In this example program, there is an error at line 2 in which the addition operator is mistakenly used instead of the subtraction operator. In cases where inputted value y is 0, the defined value of a at line 2 will be correct regardless of the error. As a result, only test cases A and B in Figure 2.6 pass, while test cases C and D fail.

Consider failing test case C. An IVMP is identified at line 2 because changing the values of x and y respectively from 1 and 1, to 0 and 0 (which are used by test case A), will correct the program output. Also, an IVMP is identified at line 6 because changing the used value of a from 2 to 0 (which is the value of a used by test case A), will correct the output as well. Although IVMPs are found at lines 2 and 6, only one of these lines contains the actual error. The IVMP at the other line is present due to the *dependence cause* for IVMPs at multiple statements. To help distinguish between these two statements, another failing run is considered.

When considering failing test case D, an IVMP is identified at line 2 because changing x and y from 0 and 1, to -1 and 0 (used by test case B), will correct the output. Also, an IVMP is identified at line 4 because changing the value of a here from 1 to -1 (the value of a in test case B) will correct the output. Here, IVMPs are found at lines 2 and 4.

```
1:  /* let (x,y) be input values */
2:  a := x + y;   /* should be x − y */
3:  if (x < y)
4:     write(a);
5:  else
6:     write(a + 1);
```

Test Case	Input Values	Actual Output	Expected Output	Result
A	$(x,y) = (0,0)$	1	1	PASS
B	$(x,y) = (-1,0)$	-1	-1	PASS
C	$(x,y) = (1,1)$	3	1	FAIL
D	$(x,y) = (0,1)$	1	-1	FAIL

Figure 2.6: Example to motivate the need to consider multiple failing runs.

Consider the statements with IVMPs in both failing runs C and D. Line 2 is associated with IVMPs in both failing runs, whereas lines 4 and 6 are associated with IVMPs in only one failing run each. Therefore, line 2 is more likely to be faulty than either lines 4 or 6.

The example from Figure 2.6 shows the benefit of considering IVMPs from multiple failing runs when ranking program statements using IVMPs.

2.4 Ranking Statements using IVMPs

Given a faulty program and a test suite containing multiple failing runs, the statements exercised by the failing runs are ranked in decreasing order of *suspiciousness* value (likelihood of being faulty). Let F be the set of all failing runs in an available test suite, and let $STMT_{IVMP}(f)$ refer to the set of all program statements associated with at least one IVMP identified from failing run f. Then the *suspiciousness of a statement* s, *suspiciousness*(s), can be defined as the number of failing runs in which at least one IVMP

was identified for that statement.

Definition 4 (Suspiciousness of a Statement s).

$$suspiciousness(s) \quad := \quad |\{f : f \in F \wedge s \in STMT_{IVMP}(f)\}|$$

Note that this definition of *suspiciousness* considers only whether or not each statement is associated with at least one IVMP. It does not account for the actual number of IVMPs associated with each statement. This is because the number of IVMPs at any given statement does not seem to be correlated with the likelihood of that statement being faulty. Instead, the number of IVMPs at a statement depends upon the structure of the statement and the number of alternate sets of values that are associated with that statement in the value profile.

Given this definition of *suspiciousness*, there can be many ties since suspiciousness values will always be whole integers in the range $[0 \cdots |F|]$, where $|F|$ is the total number of failing runs. Thus, to break ties, a prior technique for locating errors is used. This technique, called *Tarantula* [75], also computes a suspiciousness value for each statement. Tarantula is a statistical technique that relies on the following intuition: a statement is more likely to contain an error if it is exercised more often by failing runs than by passing runs. Specifically, the $suspiciousness_{tarantula}$ of a statement s is defined [74] as follows.

Definition 5 (Suspiciousness$_{\textbf{tarantula}}$ of a Statement s).

$$suspiciousness_{tarantula}(s) \quad := \quad \frac{\frac{failed(s)}{totalFailed}}{\frac{passed(s)}{totalPassed} + \frac{failed(s)}{totalFailed}}$$

In this equation, the variables $failed(s)$ and $passed(s)$ respectively refer to the number of failing and passing runs exercising statement s. The variables $totalFailed$ and $totalPassed$ respectively refer to the total number of failing and passing runs (test cases).

Tarantula was selected as the method of breaking ties for several reasons. First, computing suspiciousness values is very quick because the technique considers only statement coverage information. Second, Tarantula has been shown [74] to be more effective in locating errors on the Siemens benchmarks [64] than either *cause transitions* [22] or *nearest neighbor* [119]. Finally, Tarantula is complementary to Value Replacement: Tarantula considers statement coverage information from failing and passing tests, whereas Value Replacement looks for statements that can be shown (through IVMPs) to be able to correct the output of failing runs.

The overall Value Replacement technique for ranking program statements using IVMPs is composed of two main steps. First, multiple failing runs are searched for IVMPs. Second, statements are ranked in decreasing order of *suspiciousness*, with ties broken in decreasing order of $suspiciousness_{tarantula}$.

Figure 2.7 shows the complete Value Replacement technique. First, the value profile is constructed from the provided test suite (line 1). Next, for each failing test case in the available test suite (line 2), the set of statement instances in the execution of that failing run, with respect to the faulty program, is identified (line 3). At each statement instance (line 4), the technique identifies the set of alternate value sets from the value profile that are associated with the relevant statement (lines 5–6). Then, for each of these alternate value sets, the value set is applied to the failing run (using a value replacement) to see whether the output of the failing run changes to become correct; if so, then an IVMP is identified and reported (lines 7–9). After identifying the IVMPs for all available failing test case executions, the statements exercised by the failing runs are ranked and reported (lines 10–15). Note that only those statements exercised by failing runs need to be ranked,

input:
Faulty program P, and test suite T containing a set F of failing runs.
output:
A ranked list of statements exercised by test cases in F.
algorithm ValueReplacementRank
begin
Step 1: [Compute IVMPs for each test case in F]
1: $valProf$:= construct value profile for P with respect to T;
2: **for each** test case $f \in F$ **do**
3: $trace_f$:= statement instances executed by f on P;
4: **for each** statement instance i in $trace_f$ **do**
5: s := the statement associated with instance i;
6: $altValSet$:= alternate value sets for s in $valProf$;
7: **for each** alternate value set $v \in altValSet$ **do**
8: **if** applying v at i corrects f's output **then**
9: **report** an IVMP found at statement s in f;
endfor (each alternate value set)
endfor (each statement instance)
endfor (each failing run)
Step 2: [Use IVMPs to rank program statements]
10: $stmts$:= set of statements exercised by test cases in F;
11: **for each** statement $s \in stmts$ **do**
12: compute $suspiciousness(s)$;
13: compute $suspiciousness_{tarantula}(s)$;
endfor
14: $stmts_{ranked}$:= sort $stmts$ by decreasing $suspiciousness$, break ties by decreasing $suspiciousness_{tarantula}$;
15: **output** $stmts_{ranked}$;
end ValueReplacementRank

Figure 2.7: The Value Replacement technique.

because only these statements can possibly contain an error that could have caused one or more of the test cases to fail.

The runtime of this algorithm is bounded by the total number of program executions required to perform value replacements to identify IVMPs. This is bounded by $O(f \times t \times m)$, where f is the number of failing runs, t is the size of the longest failing execution trace, and m is the maximum number of alternate value sets to apply at any given statement. In Chapter 4, a method to reduce this runtime is developed.

2.5 Effectiveness of Value Replacement

2.5.1 Setup for Experiments

Implementation Details

The implementation uses the *Valgrind* infrastructure for dynamic binary translation [55, 103]. This system provides a synthetic CPU in software and allows for dynamic binary instrumentation of an executing program. Valgrind comes with a set of tools to perform tasks such as debugging and profiling, but new tools were created to record definition/use tracing information and to perform value replacements. Valgrind allows for instrumentation at the granularity of machine code instructions, so the implementation records traces in terms of instruction instances, and performs value replacements at the binary instruction level. Instructions are then mapped back to their corresponding statements (source code line numbers) when necessary to compute a ranked list of program statements. Note that when an alternate set of values is applied at an instruction instance in the implementation, the original values are actually overwritten in their respective memory or register locations.

As a result, any subsequent uses of these locations in subsequent instructions will involve the new values. In other words, the implementation ensures that the state of an executing program is properly modified at the point of a value replacement during execution. Also, in the experiments, several techniques are developed (described later in Chapter 4) that significantly reduce the search space for identifying IVMPs, while still allowing for highly effective results for locating errors in the benchmark programs. The experiments were run on a Dell PowerEdge 1900 server with two Intel Xeon quad-core processors at 3.00 GHz, and 16 GB of RAM.

Subject Programs and Test Suites

The Siemens suite programs [64] listed in Table 2.1 are used for the experiments. From left to right, the columns in this table show the program name, the number of lines of code, the number of provided faulty versions (each containing a seeded error), the average number of test cases in each created test suite (in parentheses, the total number of test cases available in the provided test case pools), and a brief description of the program functionality. The Siemens suite programs, along with their corresponding faulty versions and test case pools, were obtained from the *Software-artifact Infrastructure Repository* [58], organized by researchers at the University of Nebraska – Lincoln.

All faulty versions contain seeded errors. These errors are related to computation of non-address values, including errors such as operator and operand mutations, missing and extraneous code, and constant value mutations. These types of computation-related errors are distinct from memory errors, such as those that involve accessing incorrect memory locations and assigning incorrect pointer values. As a result, the implementation ignores

Program Name	**# Lines of Code**	**# Faulty Versions**	**Avg. Test Suite (Pool) Sizes**	**Program Description**
tcas	138	41	17 (1608)	altitude separation
totinfo	346	23	15 (1052)	statistic computation
sched	299	9	20 (2650)	priority scheduler
sched2	297	9	17 (2710)	priority scheduler
ptok	402	7	17 (4130)	lexical analyzer
ptok2	483	9	23 (4115)	lexical analyzer
replace	516	31	29 (5542)	pattern substituter

Table 2.1: The Siemens benchmark programs.

address values that are involved in each program execution.

Most faulty versions are seeded with a single error in a single statement, but some faulty versions involve modifications to several statements. A few faulty versions were excluded because they did not yield any failing test cases from the provided test case pools. One of the faulty versions from program `ptok2` was also excluded because the error in this case caused execution to loop for a very long time, causing traces to be very long and the Valgrind-based implementation to run out of memory.

The `tcas` program contains no loops and represents one big conditional check spread across several functions; it takes as input a set of integer parameters and reports one of three output values (or an error message if too few input arguments are specified). Program `totinfo` reads a collection of numeric data tables as input and computes statistics for each table as well as across all tables. Programs `sched` and `sched2` are priority schedulers for processes; these programs take as input a number of processes of various priorities as well as a list of scheduling commands, and outputs the processes as they complete in priority order. Programs `ptok` and `ptok2` are lexical analyzers; they tokenize an inputted character stream into a list of corresponding tokens. Program `replace` performs pattern substitution; it takes as input a source pattern, destination pattern, and character stream, and replaces

all instances of the source pattern in the character stream with the destination pattern.

For each faulty version of each program, a branch coverage adequate test suite was created by selecting test cases from the provided test case pools. To do this, a test case from the associated test case pool was randomly selected as long as it increased the cumulative branch coverage of the test cases in the suite selected so far. This process was repeated until the created test suite achieved the same level of branch coverage as the entire test case pool. It was ensured that each created test suite contained at least 5 test cases that failed and at least 5 test cases that passed (if available), to ensure a mix of failing and successful test cases in each suite.

Techniques and Metric for Comparison

The Tarantula statistical technique for ranking program statements yields the best results currently known for locating the errors of the Siemens benchmark programs [74, 75]. Thus, the main purpose of the experiments is to compare the error location effectiveness of both the Value Replacement technique (called the "IVMP" technique in these experiments) and the Tarantula technique.

1. **IVMP technique**. This is the Value Replacement technique from Figure 2.7 where ranking is based upon the *suspiciousness* formula, and ties are broken using the $suspiciousness_{tarantula}$ formula from the Tarantula technique [74]. Here, all available failing runs in the test suites are used to search for IVMPs (5 failing runs in most cases).

2. **Tarantula technique**. This technique ranks statements using only the formula for $suspiciousness_{tarantula}$, which has been shown [74] to be quite effective and provides

the best overall error location results currently known using the Siemens benchmarks. Here, the statement coverage information from all passing and failing runs in the test suites are used to rank program statements.

For comparison with the above two main techniques, statements are also ranked according to the following variations.

3. **Tarantula-Pool technique**. This is the same as the Tarantula technique, but here, each test suite is considered to be the entire test case pool (rather than the much smaller branch-adequate test suites used in the Tarantula technique). This is to study whether Tarantula is more effective when larger test suites are used.

4. **IVMP-1 technique**. This is the same as the IVMP technique, but here, statements are ranked by considering only one arbitrarily-chosen failing run when searching for IVMPs in each test suite (rather than by considering all failing runs in the suite as is done by the regular IVMP technique). This is to study the effectiveness of the technique when multiple failing runs are not considered.

5. **IVMP-2 technique**. This is the same as the IVMP technique, but here, statements are ranked by considering just two arbitrarily-chosen failing runs when searching for IVMPs in each test suite.

In the experiments, only those program statements that are executed by failing runs are ranked according to each of the above techniques. To evaluate the ranking results for each technique, a *score* is assigned to each ranked list of statements. This score represents the percentage of program statements executed by failing runs in the test suite that *need*

not be examined, if statements are examined in rank order until a faulty statement is found. Suppose that for a ranked list of statements L, the faulty statement occurs at rank r and there are a total of $totalStmtsEx$ total statements exercised by failing runs in the test suite. Then the score of ranked statement list L can be defined as follows.

Definition 6 (Score of a Ranked Statement List L).

$$score(L) \quad := \quad \frac{totalStmtsEx - r}{totalStmtsEx} \times 100\%$$

A rank value r of 1 means that the faulty statement is the first statement in the ranked list and there are no ties (the ideal situation). In the event that multiple statements are tied for a particular rank, all tied statements are given a rank value equal to the maximum rank value from among the tied statements. For example, if there are 5 statements tied for highest rank, then all 5 of them are given rank 5. This allows for the conservative assumption that all tied statements would have to be examined before any faulty statement within that tied set can be found. Intuitively, a higher score is preferable because it means that more of the statements executed by failing runs can be ignored before the faulty statement is found.

There are a few special considerations that are made in the experiments for certain kinds of errors. First, errors in constant assignment statements (15 out of a total of 129 faulty versions) will not result in any IVMPs at precisely those constant assignments (as previously explained in Section 2.2.2). However, IVMPs can be found at the statements using those defined constants. Therefore, a constant assignment error is considered to be located if one examines either the statement where it is defined (possible in the Tarantula technique only), or else a statement where that constant value is used (possible in either the Value Replacement or Tarantula techniques). Also, errors that involve omitted statements

(16 out of 129 faulty versions) imply that the statements that are missing cannot actually be located. However, statements can be located that are adjacent to the location where the code is missing, including those statements that would have influenced or would have been influenced by the missing code.

2.5.2 Effectiveness Results and Discussion

Experimental results are shown for each of the statement ranking techniques in Tables 2.2 and 2.3, and Figure 2.8. Table 2.2 shows the number (and percentage) of faulty versions with associated ranked lists of statements in each specified score range, for both the basic IVMP and Tarantula ranking techniques. Table 2.3 shows the results for each of the three techniques that are variations of the basic techniques. Figure 2.8 shows a graphical view of this data. In the graph, the x-axis represents the lower bound of each score range, and the y-axis represents the percentage of faulty versions achieving a score greater than or equal to that lower bound.

In the results, percentages are computed with respect to 129 faulty versions from among the Siemens programs. This presentation of data follows the convention of Jones et al. [74]. However, whereas [74] computes scores with respect to the total number of program statements, scores are computed here with respect to the total number of statements exercised by failing test cases in the suite. This is because statements that are not exercised by any failing test cases can be safely ignored when trying to locate the corresponding errors.

Score	Tarantula Technique	IVMP Technique
99-100%	5 (3.88%)	23 (17.83%)
90-99%	31 (24.03%)	66 (51.16%)
80-90%	24 (18.60%)	13 (10.08%)
70-80%	11 (8.53%)	9 (6.98%)
60-70%	14 (10.85%)	2 (1.55%)
50-60%	13 (10.08%)	2 (1.55%)
40-50%	3 (2.33%)	4 (3.10%)
30-40%	5 (3.88%)	4 (3.10%)
20-30%	5 (3.88%)	2 (1.55%)
10-20%	4 (3.10%)	1 (0.78%)
0-10%	14 (10.85%)	3 (2.33%)

Table 2.2: Number/score of ranked statement lists for basic techniques.

Score	Tarantula-Pool Technique	IVMP-1 Technique	IVMP-2 Technique
99-100%	7 (5.43%)	18 (13.95%)	21 (16.28%)
90-99%	41 (31.78%)	57 (44.19%)	63 (48.84%)
80-90%	22 (17.05%)	21 (16.28%)	15 (11.63%)
70-80%	7 (5.43%)	6 (4.65%)	9 (6.98%)
60-70%	16 (12.40%)	10 (7.75%)	4 (3.10%)
50-60%	11 (8.53%)	2 (1.55%)	2 (1.55%)
40-50%	3 (2.33%)	4 (3.10%)	4 (3.10%)
30-40%	5 (3.88%)	4 (3.10%)	4 (3.10%)
20-30%	3 (2.33%)	2 (1.55%)	2 (1.55%)
10-20%	6 (4.65%)	2 (1.55%)	2 (1.55%)
0-10%	8 (6.20%)	3 (2.33%)	3 (2.33%)

Table 2.3: Number/score of ranked statement lists for variation techniques.

IVMP Technique versus Tarantula Technique

The data shows that the IVMP technique overall performs much better than the Tarantula technique. Almost 18% of the faulty versions analyzed had a score of 99% or higher with the IVMP technique, whereas the same was true for only about 4% of the faulty versions using Tarantula. Similarly, almost 70% of faulty versions had a score of 90% or higher using the IVMP technique, while the same was true for about 28% of the faulty versions using Tarantula. Among all 129 faulty versions, the IVMP technique was able to

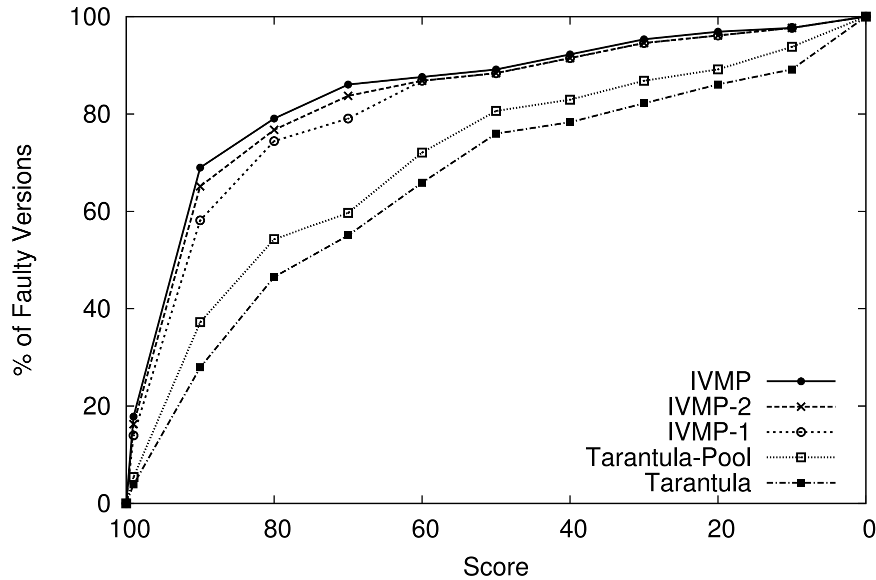

Figure 2.8: Comparison of statement ranking techniques.

uniquely identify the faulty statement (assign it rank 1) in 39 cases. Tarantula was able to do so in only 5 cases. Note that even though the IVMP technique was able to uniquely identify the faulty statement in 39 cases, only 23 cases yielded scores of 99% or more. This is because in the `tcas` program, the number of statements exercised by failing test cases was small enough that even a rank of 1 would lead to a score less than 99%.

Out of 129 faulty versions in the experiments, there were only 16 cases where the IVMP technique assigned a lower rank to a faulty statement than Tarantula. These cases occurred where IVMPs were found at non-faulty statements in more failing runs than at faulty statements, giving the non-faulty statements higher rank. However, in many of these cases, the non-faulty statements with higher rank were still near to the faulty statements via static dependence edges (recall the *dependence cause* for IVMPs at multiple statements described previously in Section 2.3). In 18 faulty versions, the IVMP technique

and Tarantula gave the faulty statement identical ranks. In some of these cases, this was due to finding no IVMPs in any failing runs, causing *suspiciousness* values to be identical and ranking to be done solely by breaking ties using $suspiciousness_{tarantula}$. In the remainder of the cases (95 of them), the IVMP technique gave the faulty statement higher rank than Tarantula, due to IVMPs being found at the faulty statement.

Comparison with Other Variation Techniques

The results for the Tarantula-Pool technique indicate that fault localization is indeed more effective for Tarantula when larger test suites are used. However, Tarantula-Pool is still considerably less effective overall on the Siemens benchmarks than the IVMP technique. In fact, Tarantula-Pool is also considerably less effective than either the IVMP-1 or the IVMP-2 technique, which both consider fewer failing test cases when searching for IVMPs than the regular IVMP technique. However, as might be expected, IVMP-2 is slightly less effective overall than the IVMP technique, while the IVMP-1 technique is also slightly less effective than the IVMP-2 technique. Thus, the IVMP technique is generally more effective as more failing runs are considered, but even when considering just a single failing run, the IVMP-1 technique is still more effective than the Tarantula-Pool technique that considers statistical coverage information taken from very large test suites.

Other Observations

Program `totinfo` is an unusual case among the Siemens programs. For this particular benchmark program, only 8 faulty versions resulted in the IVMP technique performing better than Tarantula, while 5 cases had the IVMP technique performing worse, and 10

cases had both techniques performing equally well. These results were generally not as good as the results from the other Siemens programs, in which the IVMP technique usually performed much better as compared to Tarantula. It was discovered that for `totinfo`, relatively few IVMPs were found as compared to the other Siemens programs. This is due to the fact that `totinfo` performs floating-point computations and outputs floating-point values. Thus, it is very difficult to cause output in `totinfo` to change to become precisely correct when value replacements are performed.

Another observation is that the sizes of the value profiles for each faulty version seem to increase logarithmically as the number of test cases in the suites increase. To study this in more detail, value profiles were constructed for five arbitrarily-chosen faulty versions from each Siemens program using test cases from the available test case pools. The results for each faulty version in a subject program were then averaged and plotted as shown in Figure 2.9.

As can be seen in the figure, the curves for most benchmark programs become nearly horizontal over time as more test cases are considered in the value profile. One notable exception is for program `totinfo`, which has a curve that increases much higher than that of all the other programs. This is because `totinfo` uses many floating-point values, which are highly likely to be different from test case to test case.

2.5.3 Experiments with Larger Benchmark Programs

Some additional experiments were conducted to see whether Value Replacement may still yield effective results on larger subject programs. The analyzed programs are described in Table 2.4, and were obtained from the *Software-artifact Infrastructure Repos-*

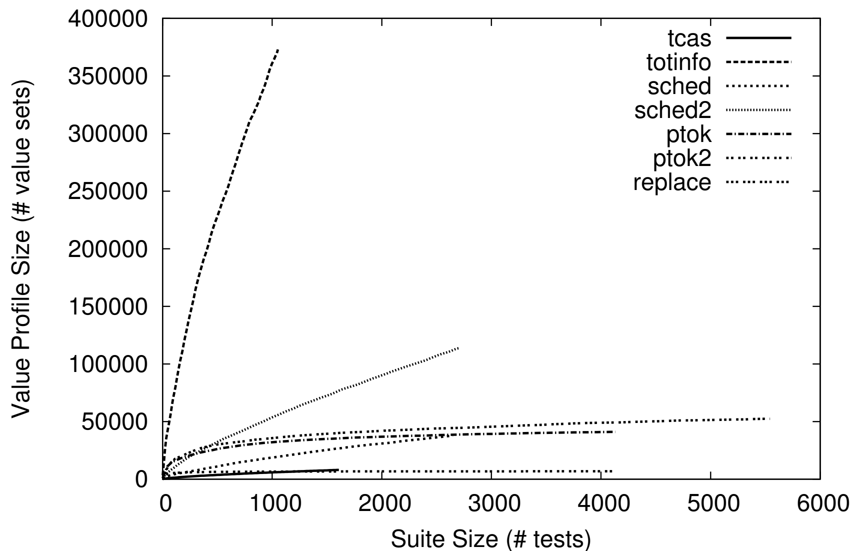

Figure 2.9: Increase in value profile size as suite sizes increase.

itory [58]. One error was selected from each program to locate. Program `space` contains a known error in which a condition ($error\ != 0$) should instead be ($error == 17$). The `grep` program contains a known error in which using parameters $-i$ and $-o$ simultaneously may lead to incorrect output. Programs `sed`, `flex`, and `gzip` contain seeded errors [58] of the following respective types: a "-1" term is missing from an expression; command-line parameters are incorrectly processed; and input files with improper file names are incorrectly processed.

These experiments followed a similar setup as for the Siemens benchmark pro-

Program Name	# Lines of Code	Error Type	Program Description
space	6.2 K	real	ADL interpreter
grep-2.5	5.8 K	real	pattern matcher
sed-4.1.5	13.0 K	seeded	stream editor
flex-2.5.1	10.0 K	seeded	lexical analyzer generator
gzip-1.3	5.2 K	seeded	file compressor

Table 2.4: Larger benchmark programs.

Program Name	Faulty Statement Rank	
	Tarantula technique	IVMP technique
space	106	5
grep-2.5	213	3
sed-4.1.5	35	3
flex-2.5.1	45	1
gzip-1.3	96	1

Table 2.5: Experimental results using the larger benchmark programs.

grams, except here, branch-coverage adequate test suites were not created since the programs are much larger. Instead, each suite consists of a few failing runs and several $(5-6)$ passing runs. The experimental results using these larger programs are shown in Table 2.5. For each program, the rank of the faulty statement is shown for each of the Tarantula and IVMP techniques. From these results, it can be seen that the IVMP technique is able to take advantage of IVMPs to demonstrate significant improvements in error location effectiveness over Tarantula.

2.6 Summary

In this chapter, the state alteration technique called Value Replacement was developed for assisting in the task of locating software errors. This technique repeatedly alters the state of failing executions by performing value replacements to identify IVMPs. These IVMPs show how values used at particular program statements can be altered so that failing runs instead produce correct output. Using these IVMPs, executed statements can be ranked according to their likelihood of being faulty. Experimental results show that for the benchmark programs studied, Value Replacement can produce ranked lists of statements that are generally very effective at quickly leading a developer to a faulty statement. More-

over, the technique was seen to be more effective than a prior error location technique that had yielded the best results previously known for the benchmark programs in the study. However, there are several questions that need to be answered regarding Value Replacement.

How to handle multiple simultaneous errors? Some of the Siemens faulty versions contain multiple errors. In these cases, Value Replacement was still able to find IVMPs for at least one of the errors and achieve statement ranking results that locate the error. However, in general the presence of multiple simultaneous errors can cause test cases to fail due to different errors (or different combinations of errors). Because of the way suspiciousness values are computed in the Value Replacement technique, this can lead to decreased relative suspiciousness of the statements that are truly faulty, thereby diminishing the effectiveness of the technique. In the next chapter, techniques are developed that generalize Value Replacement to handle the situation of multiple simultaneous errors.

How to improve scalability? Although Value Replacement was shown to be considerably more effective than Tarantula in locating the errors from the Siemens benchmark programs, it is also true that Value Replacement requires considerably more computation time than Tarantula. One of the major questions about Value Replacement is whether it can scale to large programs in general. Note that Value Replacement is not limited by program size, but rather, by execution trace length. Recall that the search for IVMPs occurs at every statement instance in a failing run using different sets of alternate values. For very long failing runs, Value Replacement cannot scale unless techniques are used to limit the search space for IVMPs. In Chapter 4, techniques are developed to drastically reduce the IVMP search space and improve the efficiency of implementation, while retaining the effectiveness of Value Replacement in locating errors.

How to handle address values and memory errors? In the basic value replacement technique, address values are ignored because the Siemens benchmark programs do not involve memory-related errors, and so errors can still be effectively located in the experiments by ignoring address values. It can be observed, however, that handling address values would require special consideration in Value Replacement. This is because address values from different test case executions cannot simply be blindly substituted into a particular failing run when performing value replacements, because address values are execution-specific and have no meaning outside of a given execution. As a result, without further enhancements, Value Replacement may have limited effectiveness in locating memory errors. One approach would be to enhance the Value Replacement algorithm to properly consider address values. However, memory errors are unique in that they often involve *memory corruption*, which can lead to program crashes. Chapter 5 instead develops a different state alteration technique called *Execution Suppression* that focuses on isolating memory corruption to locate memory errors. Execution Suppression is more efficient for locating memory errors than Value Replacement, because while Value Replacement aggressively performs many blind state alterations to search for IVMPs, Execution Suppression performs targeted state alteration using information that is uniquely revealed by memory failures.

Chapter 3

Value Replacement and Multiple Simultaneous Errors

In the previous chapter, the Value Replacement state alteration technique for locating software errors was developed. This technique uses multiple failing runs in order to rank program statements according to likelihood of being faulty. However, the technique implicitly assumes that all considered failing runs fail due to the same error. In the event that multiple simultaneous errors are present in software, different failing runs may fail due to different errors, or different combinations of errors. This can decrease the perceived suspiciousness of the faulty statements as compared to the non-faulty statements, thereby making it more difficult to isolate the faulty statements and decreasing the effectiveness of the technique for locating errors.

This chapter shows how to generalize the Value Replacement technique into an iterative technique that can effectively handle the situation when multiple errors are present in software [69]. The goal is to present a ranked list of program statements to the developer

to isolate each individual error; each time a located error is fixed, then another ranked list is presented to the developer as long as at least one test case still fails. Three variations of this idea are developed that involve different techniques for computing a new ranked list of statements on each iteration.

- First, a Minimal-Computation technique is developed in which Value Replacement is applied only once to rank program statements, and the search for all errors is performed within the single ranked list (i.e., the same ranked list is reported to the developer on each iteration). This simple technique has relatively low cost as compared to the other techniques that will be described, because it performs Value Replacement only once to locate all errors. However, the effectiveness of this technique is relatively low because only a single ranked list is used to locate all errors; the ranked list is never updated to account for revised dynamic information that results from fixing an error.

- Second, a Full-Recomputation technique is developed in which Value Replacement is iteratively invoked to find and fix one error at a time. This technique is highly effective as compared to the other techniques, because it computes a new ranked list to locate each error. Each ranked list is computed using the updated dynamic information that results from fixing the previously-located error. However, this technique also incurs relatively high cost, because the full Value Replacement technique must be invoked on each iteration.

- Finally, a middle-ground, Partial-Recomputation technique is developed in which only a part of the required computation for Value Replacement is performed on each itera-

tion to find and fix an error. This technique incurs less cost than Full-Recomputation, while retaining much of its effectiveness.

3.1 Techniques to Locate Multiple Errors

Figure 3.1 depicts an overview of the core Value Replacement technique (A) that is used to locate single errors, and three variations of the technique (B–D) that can be used to iteratively locate multiple errors. In the core technique (Figure 3.1 (A)), a faulty program and associated test suite are passed as input to the Value Replacement algorithm (described in Chapter 2), which computes a ranked list of program statements that can be examined by a developer in order to find and fix the (single) error. To handle multiple errors, Figure 3.1 (B) illustrates the Minimal-Computation technique in which Value Replacement is performed only once, and the developer uses the single ranked list of statements to search for faulty statements as needed. Figure 3.1 (C) shows the opposite extreme: the Full-Recomputation technique in which Value Replacement is invoked to allow a developer to find and fix an error, then this process is fully repeated on the new version of the program as necessary until all errors are fixed. Figure 3.1 (D) illustrates the Partial-Recomputation technique that performs only partial Value Replacement computation on each iteration. The three techniques (B–D) for handling multiple simultaneous errors are now described in detail.

3.1.1 Minimal-Computation Technique

The algorithm for the Minimal-Computation technique is presented in Figure 3.2. The technique takes as input a faulty program and a corresponding set of test cases contain-

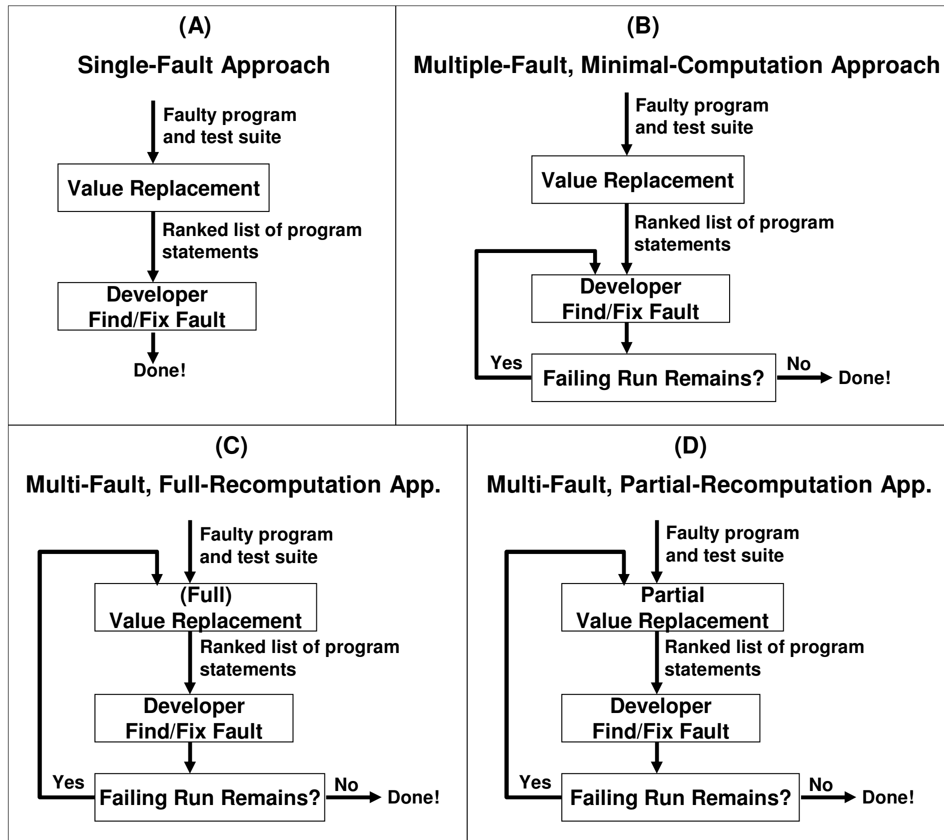

Figure 3.1: Single-fault core technique (A) and multi-fault generalized techniques (B-D).

ing at least one failing run. First, the Value Replacement technique is executed to obtain a ranked list of program statements (line 1). Then, as long as a failing run exists in the test suite with respect to the current version of the program (line 3), the ranked list is used to locate an error in the program (line 4), which is then fixed (line 5), resulting in a new version of the program with the error removed. The actual identification and fixing of a faulty statement is done manually by a developer. If at least one run still fails on the new version of the program, then the (same) ranked list is consulted again to find and fix another error (back to line 3). Under this technique, the computation time is expected to be comparable to that of the single-fault core Value Replacement technique, since a ranked list of program statements need be computed only once. However, the drawback is that the

```
input:
    Faulty program P, and test suite T containing at least one failing run.
output:
    Program P' such that no tests in T fail.
algorithm MinimalComputationTechnique
begin
1:  rankedList := Run_Value_Replacement(P, T);
2:  P' := P;
3:  while ∃ a failing run in T with respect to P' do
4:      E := next error located using rankedList;
5:      P' := version of P after fixing E;
    endwhile
end MinimalComputationTechnique
```

Figure 3.2: Minimal-Computation technique to locate multiple errors.

average effectiveness to locate each individual error may be reduced, since the ranked list is never updated as the program is modified and errors are corrected over time.

3.1.2 Full-Recomputation Technique

The algorithm for the Full-Recomputation technique is presented in Figure 3.3. This technique is identical to the Minimal-Computation technique, except the invocation of the Value Replacement technique has been moved to *inside* the main loop (line 1 in Figure 3.2 is now line 3 in Figure 3.3). The effect is that a revised ranked list is computed on each iteration when an error is found and fixed. This ensures that the technique has up-to-date data that can be used to compute a more effective ranking on each iteration. However, the timing requirements increase significantly because Value Replacement must be invoked on every iteration to search for new IVMPs.

```
input:
    Faulty program P, and test suite T containing at least one failing run.
output:
    Program P' such that no tests in T fail.
algorithm FullRecomputationTechnique
begin
1:  P' := P;
2:  while ∃ a failing run in T with respect to P' do
3:     rankedList := Run_Value_Replacement(P', T);
4:     E := next error located using rankedList;
5:     P' := version of P after fixing E;
    endwhile
end FullRecomputationTechnique
```

Figure 3.3: Full-Recomputation Technique to locate multiple errors.

3.1.3 Partial-Recomputation Technique

The algorithm for the Partial-Recomputation technique is presented in Figure 3.4. This technique consists of two main steps. In the first step, a *set of ranked lists* is computed, and these lists are used to find and fix a first error in the program. In the second step, the technique iteratively performs partial Value Replacement re-computation, updates any affected ranked lists, and then uses the revised ranked lists to find and fix a next error.

Step 1: Initialize ranked lists and locate the first error (lines 1-10). In this step, the approach first collects together all statements exercised by failing runs to consider for ranking purposes (line 2). Next, for each of these statements s, a ranked list of program statements is computed using Value Replacement by searching for IVMPs in only those failing runs exercising s (lines 3-6). The intuition for this step is as follows: it is known that at least one of the statements s is faulty, and maximum suspiciousness is most likely to be achieved for such a statement if statements are ranked based on the IVMP information of only those failing runs which exercise s. Since the faulty statements are not known, a

```
input:
    Faulty program P, and test suite T containing at least one failing run.
output:
    Program P' such that no tests in T fail.
algorithm PartialRecomputationTechnique
begin
Step 1: [Compute ranked lists and find/fix a first error]
1:  RankedLists := {};
2:  Stmt_fail := statements exercised by failing runs in T;
3:  for each statement s ∈ Stmt_fail do
4:     F := failing runs in T that do not exercise s;
5:     rankList := Run_Value_Replacement(P, T − F);
6:     RankedLists := RankedLists ∪ {rankList};
    endfor
7:  selectedList := removeFirstList(RankedLists);
8:  E := next error located using selectedList;
9:  faultyStmt := the statement containing E;
10: P' := version of P after fixing E;
Step 2: [Iteratively revise ranked lists and find/fix remaining errors]
11: while ∃ a failing run in T with respect to P' do
12:    Fail := failing runs in T exercising faultyStmt with respect to P';
13:    compute IVMPs for each run in Fail with respect to P';
14:    update any affected lists in RankedLists;
15:    selectedList := removeNextList(RankedLists);
16:    E := next error located using selectedList;
17:    faultyStmt := the statement containing E;
18:    P' := version of P after fixing E;
    endwhile
end PartialRecomputationTechnique
```

Figure 3.4: Partial-Recomputation technique to locate multiple errors.

ranked list is computed for each considered statement. Note that overall, this step requires a search for IVMPs in each failing run at most once; the IVMP information of a failing run can simply be looked up when needed, if a search for IVMPs was previously conducted in that run.

Next, from among the computed set of ranked lists, the technique selects and removes one of the lists to be used to locate the first error (line 7). This is determined

by choosing the list in which the statement at the front of the ranked list has highest suspiciousness value. In the event of a tie among ranked lists, then subsequent statements in the tied lists are examined as necessary to break the tie. In this way, the selected list is most likely to quickly lead a developer to the first error. Thus, the developer uses the selected ranked list to find the first error (lines 8-9) and fix it (line 10).

Step 2: Iteratively revise ranked lists and locate the remaining errors (lines 11-18). The second step iterates as long as a failing run still exists (line 11). First, in the new version of the program, the technique identifies the set of failing runs exercising the faulty statement that was just fixed (line 12). For only these failing runs (not for all failing runs), IVMPs are re-computed (line 13). Note that this step only searches for IVMPs in a subset of failing runs; it does not actually compute suspiciousness values and a ranked list as is done in the call to Run_Value_Replacement at line 5. Importantly, note that this step will not necessarily re-compute IVMPs for all runs that still fail (like what is done in the Full-Recomputation technique). This is how Partial-Recomputation can incur less total cost than Full-Recomputation.

Based on the updated IVMP information, any affected ranked lists from the set of maintained lists are updated so that their rankings may change (line 14). A ranked list is considered affected if it was computed using one of the failing runs for which new IVMPs were just identified.

Next, using the revised set of ranked lists, the next one to use to locate the next error is selected and removed (line 15). Note that the selection criterion here is different than in line 7. In this case, the technique selects the ranked list for which the statements near the front of the list are the most *different* from those statements near the fronts of the

previously-selected lists. This is computed by setting a difference threshold value D, and then scanning all ranked lists in order in parallel, selecting the first ranked list that achieves D different statements as compared to those statements in the previously-selected lists (a value of $D = 10$ was used in the experiments since that led to effective error location results on the benchmark programs). The intuition for this criterion is that a different error is likely to have a different set of statements with high suspiciousness, than for those errors already found and fixed. The selected list is finally used to locate (lines 16-17) and fix (line 18) the next error. This process then iterates again if any failing runs still remain (back to line 12).

Example

The Partial-Recomputation technique is demonstrated with an example. Figure 3.5 shows an example control-flow graph of a program containing 5 statements, two of which happen to be faulty. Suppose a test suite is available that contains 3 failing runs as depicted in the figure, with associated execution traces and sets of statements containing IVMPs as shown. In this case, two of the runs fail due to faulty statement 2, and one of them fails due to faulty statement 4. In the first step, the set of statements exercised by failing runs is identified (all statements in this case). Next, a ranked list of program statements is computed and associated with each one of these statements, by ordering statements according to suspiciousness value. Recall that the suspiciousness value is the number of (considered) failing runs in which the associated statement has an IVMP. The 5 computed ranked lists for the example are shown in Figure 3.5. Each of these ranked lists is computed using the IVMP information from only those failing runs exercising the statement that is

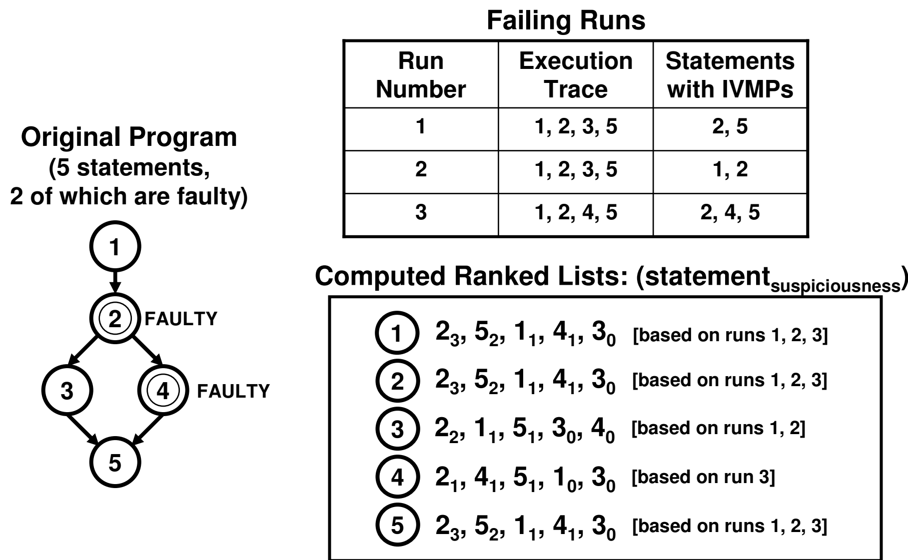

Figure 3.5: Abstract example for the Partial-Recomputation technique, part 1 of 2.

associated with the ranked list. For example, the ranked list associated with statement 4 is computed using only run 3 (since only run 3 exercises statement 4).

Next, the technique identifies the first ranked list (from among the 5 computed lists) to remove and report to a developer. This is the one with highest suspiciousness values at the front of the list. Ranked lists 1, 2, and 5 have the first ranked element with highest suspiciousness. However, since these lists happen to be identical (no ties can be broken), an arbitrary choice is made from these lists. Suppose list 1 is selected, removed, and reported to the developer. Then faulty statement 2 is immediately identified because it occurs at the front of the selected list. The developer can then fix this faulty statement.

Figure 3.6 shows how the situation might look after faulty statement 2 is fixed. In this case, statement 4 is the only remaining faulty statement. Assume that run 3 is the only run that still fails. Further assume that on the new version of the program, run 3 is associated with an IVMP at only statement 4. Next, the second main step of the

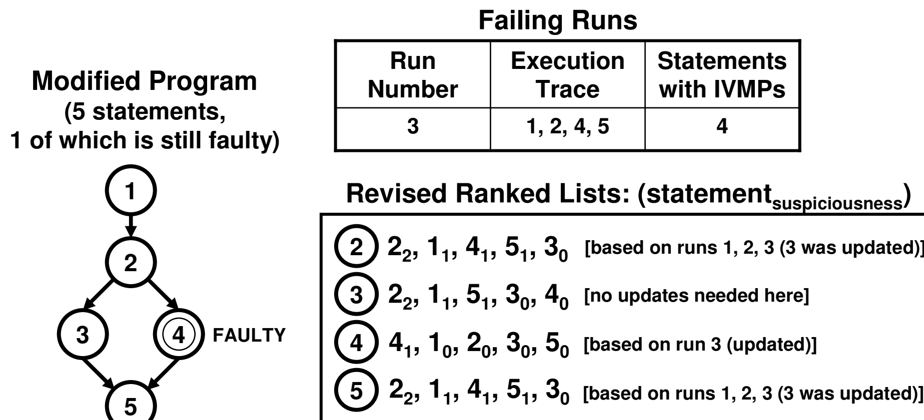

Figure 3.6: Abstract example for the Partial-Recomputation technique, part 2 of 2.

Partial-Recomputation technique is executed. First, the approach identifies the subset of newly-failing runs that need to be re-searched for IVMPs. In the example, failing run 3 exercises statement 2 (the most recently-fixed statement), so run 3 must be re-searched for IVMPs. In practice, not all failing runs may need to be re-searched for IVMPs in this step. Next, from among the remaining ranked lists, only lists 2, 4, and 5 are affected by the new IVMPs and need to be updated (list 3 was not originally computed using run 3). In the original version of the program, run 3 was associated with IVMPs at statements 2, 4, and 5. However, in the new version of the program (with corrected statement 2), run 3 is associated with IVMPs at only statement 4. Thus, ranked lists 2, 4, and 5 are updated to reflect a decrease of 1 in the suspiciousness values for statements 2 and 5 (shown in Figure 3.6). Now, the next ranked list to remove and report to the developer is selected. In this case, the technique selects the ranked list from among those remaining, that is most different from the first-selected ranked list, in terms of the elements near the fronts of the lists. Since the first-selected ranked list had started with statement 2, then from among the remaining lists, list 4 is the most different because it is the only remaining list that does

not also start with statement 2. Thus, ranked list 4 is selected. This allows the developer to immediately fix faulty statement 4 (since it appears at the front of the ranked list). At this point, no failing runs remain since all errors are fixed, and the technique terminates. Overall in this example, two ranked lists were selected and reported to the developer, each list accurately identifying one of the errors with highest suspiciousness.

3.2 Effectiveness Comparison of Techniques

3.2.1 Setup for Experiments

Implementation Details

As described in Section 2.5.1, the implementation of the core Value Replacement error location technique is based on the *Valgrind* infrastructure [55, 103] that allows for dynamic binary translation of an executing program. On top of this, `Java` was used to implement the three variation techniques that handle multiple simultaneous errors. Also, the experiments involve several implementation techniques that significantly improve the efficiency of searching for IVMPs, without actually reducing the search space; these efficiency improvements will be described in detail in Chapter 4. The experiments were conducted using a Dell PowerEdge 1900 server with two Intel Xeon quad-core processors at 3.00 GHz, with 16 GB of RAM.

Subject Programs and Test Suites

Like in the previous chapter, the experimental subjects are based on the Siemens benchmark programs and test cases [64] described previously in Table 2.1. Recall that each

Siemens benchmark program is associated with a set of faulty versions – each containing a seeded error – and a pool of test cases.

For the experiments, a set of programs containing multiple errors is required. To create a set of multiple-error faulty versions for each subject program, errors were randomly selected from among the available seeded errors to create faulty versions that each contain 5 seeded errors. It was ensured that each error in a multiple-error version is contained in a different statement. In total, up to 20 unique 5-error faulty versions for each subject program were created, as permitted by the set of available errors. Only 2 and 11 such versions could be created for programs `ptok` and `ptok2` respectively, due to a limited number of available errors, some of which conflicted by being located in the same statement and could not be incorporated into the same faulty version.

For each multiple-error version of each subject program, a test suite was created by selecting tests randomly from the associated test case pool until the following criteria were achieved: (1) the suite is statement-coverage adequate (achieving the same statement coverage as the test case pool); (2) for each faulty statement present in the multiple-error version, there exists a failing run in the suite exercising that statement; and (3) there are at least 5 failing runs and 5 passing runs in the suite (to ensure a good mix of failing and passing runs). Table 3.1 shows the number of multiple-error faulty versions created for each subject program, as well as the average test suite size associated with each one.

Techniques and Metric for Comparison

The experiments compare the effectiveness for locating errors using the following techniques that generalize Value Replacement and handle multiple simultaneous errors.

Program Name	# 5-Error Faulty Versions	Avg. Test Suite Size (# Failing Runs/# Passing Runs)
tcas	20	11 (5 / 6)
totinfo	20	22 (10 / 12)
sched	20	29 (10 / 19)
sched2	20	30 (9 / 21)
ptok	2	32 (8 / 24)
ptok2	11	29 (5 / 24)
replace	20	38 (9 / 29)

Table 3.1: Multiple-error experimental subjects.

1. **Minimal-Computation technique (MIN)**. Under this technique, the core Value Replacement technique is applied only once to obtain a single ranked list of program statements, which is then consulted as necessary until all errors are located.

2. **Full-Recomputation technique (FULL)**. Under this technique, the original Value Replacement technique is iteratively applied to locate and fix each error, one at a time.

3. **Partial-Recomputation technique (PARTIAL)**. Under this technique, multiple ranked lists of program statements are computed and iteratively revised through partial recomputation of IVMPs (from only a subset of failing runs) to locate and fix each error.

4. **The ideal situation (IDEAL)**. The "ideal" situation for finding each error is considered to be the case where that error exists in isolation in a program (with no other errors present). This situation is most likely to lead to the best location results for each error, when using Value Replacement. These "ideal" single-error results are used to compare against the above three techniques that handle multiple errors. Note that this definition of "ideal" is given with respect to the Value Replacement technique locating single errors that are present in isolation. This definition is different than the

more general notion of "ideal" results for error location, in which a faulty statement is uniquely given highest suspiciousness.

To compare the ranking results of each technique, a *score* is assigned to each ranked list of statements as was previously described in Section 2.5.1.

3.2.2 Effectiveness Results and Discussion

Table 3.2 shows the average score values achieved for each located error (from among all individual errors contained within the multiple-error versions associated with each benchmark program). As shown in the table, the FULL approach is able to achieve average score values that are very close to the IDEAL values in most cases (within one or two percentage points). The exceptions are programs `sched2`, `ptok`, and `ptok2`, in which the FULL approach achieves average results that are about 5% − 6% less than the IDEAL results. It was found that for these three programs that contain relatively few distinct errors (shown as the number of faulty versions in Table 2.1), there were a small number of particular errors in which IVMPs could not be found at the faulty statements, thus resulting in poor ranking results. Since these "problem" errors were repeatedly selected from a relatively small set of total errors, they were present in relatively many of the multiple-error faulty versions for these programs, negatively affecting the average results.

In all cases, the PARTIAL approach is able to achieve average score values that are within 5% of the FULL approach. In some cases, the difference is quite small. For example, in the `replace` program, the FULL approach has an average score of 86.98%, while the PARTIAL approach yields almost the same average score: 86.50%. For `ptok2`, FULL has an average score of 84.37% while PARTIAL has 84.13%. This suggests that PARTIAL may

Program Name	Average Score (%)			
	IDEAL	**FULL**	**PARTIAL**	**MIN**
tcas	83.64	82.87	77.98	68.82
totinfo	64.45	63.14	60.29	52.78
sched	88.56	88.28	85.13	84.72
sched2	64.75	58.15	56.81	56.20
ptok	76.28	70.14	65.55	59.00
ptok2	89.62	84.37	84.13	80.69
replace	88.17	86.98	86.50	76.27

Table 3.2: Average score achieved for each located error using each technique.

be effective at approximating the effectiveness of FULL in certain cases.

The MIN approach has the lowest average scores in all cases. When compared to the PARTIAL approach, the MIN results are still sometimes considerably lower. For example, in program `tcas`, the MIN approach yields an average score of 68.82%, which is about 9% less than PARTIAL, 14% less than FULL, and 15% less than IDEAL. For `replace`, MIN yields an average score that is about 10% less than that achieved by PARTIAL.

Table 3.2 suggests that if effectiveness is the primary concern when locating multiple errors in software, the FULL and PARTIAL approaches may be the best choices. However, the FULL approach may be prohibitively time-consuming in some cases. In these situations, the PARTIAL approach may be preferable to achieve better running time. More details about the runtime of these techniques are described in the next chapter.

3.2.3 Comparison to a Clustering Technique

In the paper "Debugging in Parallel" [73], a framework is described for *parallel debugging* in the presence of multiple errors. In this work, failing runs are clustered according to one of two proposed clustering techniques, and then used to create specialized test suites that are each targeted to a single error. Unlike the iterative techniques for Value Re-

placement developed in this chapter in which errors are located and fixed one-at-a-time, the clustering techniques allow for parallel workflows in which multiple errors can be debugged in parallel.

To study the effect of clustering, one of the proposed clustering techniques was implemented ("Technique 2" in [73], selected based on ease of implementation). In this clustering technique, each individual failing test case in an available test suite is used to compute a suspiciousness ranking. These suspiciousness rankings are then checked against each other for similarity using a metric called *Jaccard Set Similarity*, which is defined as the ratio of the sizes of the intersection and the union between two sets. The failing runs associated with the suspiciousness rankings that are considered "similar" to each other are then clustered together. Each cluster of failing runs is then used to compute a ranked list of program statements that targets a particular error. To determine whether two suspiciousness rankings are "similar" or not, a threshold value T (between 0 and 1) is set such that a set-similarity value greater than or equal to T is considered to be "similar".

The above clustering technique is general and can be used in conjunction with any error location technique that computes a suspiciousness ranking. An experiment was conducted to perform the above clustering technique in conjunction with the Value Replacement technique for computing suspiciousness rankings. The computed clusters were then used to compute ranked lists of program statements that could be used to locate errors. Table 3.3 shows these results, on average for each subject program. The table shows the results for different similarity threshold values T, which guide how the clusters are formed and can therefore have a significant impact on the overall results.

From this table, it can be seen that the average score values are generally signif-

Program Name	**Average Score (%)**				
	T=0.1	**T=0.3**	**T=0.5**	**T=0.7**	**T=0.9**
tcas	69.43	69.97	67.82	67.68	59.59
totinfo	54.23	56.37	56.89	56.84	54.66
sched	91.29	91.97	88.96	88.54	88.45
sched2	55.39	55.32	53.79	47.28	42.30
ptok	57.25	57.25	58.38	60.50	59.50
ptok2	80.78	80.64	80.91	78.94	78.13
replace	77.85	77.62	76.26	66.68	63.26

Table 3.3: Average score for each located error using clustering (experiments were performed for different threshold values T).

icantly lower for the clustering technique than for the FULL and PARTIAL techniques as shown previously in Table 3.2. One exception is for program `sched`, in which the clustering technique is able to achieve slightly higher average score values for the lower threshold values T.

An interesting observation is that in most of the subject programs, the average score achieved by the clustering technique tends to decrease slightly as the similarity threshold value T increases. Since the effect of increasing the T value means that it is harder for suspiciousness rankings to be marked as "similar", then a higher T implies smaller cluster sizes. Thus, larger clusters (targeted suites containing more failing runs) seem to promote more effective results overall in this experiment.

Overall, the results in Table 3.2 suggest that the techniques developed in this chapter, which are aimed specifically at improving the effectiveness of Value Replacement in the context of multiple errors, may be preferable to the more general clustering technique when applied in conjunction with Value Replacement. However, the benefit of clustering is that it allows for debugging of multiple errors in parallel, whereas the techniques proposed in this chapter are iterative in nature, meant to isolate only one error at a time.

3.3 Summary

In this chapter, three techniques were developed that generalize the core Value Replacement technique for locating errors, so that Value Replacement can perform effectively in the presence of multiple simultaneous errors in software. All three of the proposed techniques iteratively report a ranked list of program statements such that each reported list can guide a developer to some error in the program as quickly as possible. Each time an error is located and fixed, a new ranked list is computed and reported. However, the three techniques differ in they way in which they compute a ranked list of statements on each iteration. In the Minimal-Computation technique, an initial ranked list is computed, and then the same ranked list is used over and over to locate each subsequent error. In the Full-Recomputation technique, a new ranked list is fully computed on each iteration as errors are located and fixed. In the Partial-Recomputation technique, each iteration only performs partial re-computation of IVMPs to compute a new ranked list. In the experiments, it was seen that the Partial-Recomputation technique was nearly as effective as Full-Recomputation in locating errors in the benchmarks, whereas the Minimal-Computation technique was relatively less effective in general.

In Chapter 2 and in the current chapter, experimental results were presented concerning only the *effectiveness* of the proposed techniques. In the next chapter, *efficiency* issues related to Value Replacement are described, and several techniques for improving the efficiency are developed. Experimental results illustrating the efficiency of Value Replacement are also presented.

Chapter 4

Efficiency of Value Replacement

In this chapter, issues related to the efficiency of Value Replacement are considered. First, some important observations about the cost of Value Replacement are made. Then, a set of *lossy* and *lossless* techniques for improving the efficiency of Value Replacement are developed. The *lossy* techniques reduce the search space for identifying IVMPs, meaning that some IVMPs can potentially be missed. However, effective error location results can still be achieved under the reduced search space. The *lossless* techniques do not actually reduce the search space for IVMPs, but they involve other implementation enhancements that can significantly reduce the time required to search for IVMPs, as compared to a brute-force implementation that just performs each complete value replacement one at a time. Finally, an experimental evaluation of the efficiency of Value Replacement for locating single and multiple errors is presented, to complement the effectiveness evaluations already described in Chapters 2 and 3.

4.1 The Cost of Value Replacement

The Value Replacement technique involves performing multiple value replacements to try to identify the statements in a program that can be associated with IVMPs. The technique first considers multiple failing program executions. Within each failing execution, every statement instance is considered. At each statement instance, mutiple alternate sets of values are considered. For each alternate set of values, a value replacement is performed in the program execution in order to determine whether an IVMP results. Under a naive, brute-force implementation, each value replacement requires a complete program execution, from the start of execution, until the point of the value replacement, until the point of execution termination in order to examine the output. Let F be the total number of failing program executions, I be the maximum number of statement instances across all failing executions, and V be the maximum number of alternate sets of values to apply at any given statement instance. Then the total number of program executions required by Value Replacement (under a naive implementation) is bounded by $O(F \times I \times V)$.

In practice, the total number of program executions required to search a set of failing runs for IVMPs can be very large. Suppose there are only 3 failing runs to consider, with each run consisting of 1000 statement instances, and suppose there are 5 alternate sets of values to apply at each statement instance. Then the total number of program executions required to search for IVMPs is $3 \times 1000 \times 5 = 15,000$. Even if each value replacement program execution takes 0.25 seconds to perform, this still requires total runtime of over 1 hour to perform all value replacements. Moreover, in practice, execution traces are likely to be much longer. A more realistic situation might involve 5 failing executions, each with around 50,000 statement instances, and an average of 15 alternate value sets to apply at each

instance. This would require a total of $5 \times 50,000 \times 15 = 3,750,000$ program executions. Now, even if each execution still requires 0.25 seconds, then it would take over 10 *days* to perform all value replacements.

For the single-error Siemens benchmark programs from Chapter 2, full IVMP searches were actually not conducted in the experiments. Instead, a set of lossy techniques were implemented (developed in the next section) that significantly reduced the search space for IVMPs, but that still resulted in good error location results. The reason the full search was not conducted was because it would have been infeasible (without any other efficiency improvements) to locate some of the errors. Figure 4.1 shows, for each of the 129 Siemens faulty versions considered in the experiments in Chapter 2, the total number of program executions that would have been required to locate the respective error using a full IVMP search. Note that the y-axis is specified in millions. Along the x-axis, the faulty versions are specified in decreasing order of the number of required program executions. From this figure, it can be seen that nearly 20 faulty versions would have required over 5 million program executions each, with the worst case requiring over 20 million program executions. The main reason why some of these faulty versions require so many program executions to search for IVMPs, is because of certain failing execution traces that are very long. Even though the Siemens programs are relatively small in terms of the number of lines of code, certain inputs to several of these programs can still cause the execution traces to be very long.

Under a naive, brute-force implementation, Value Replacement cannot scale to handle long execution traces. Thus, several techniques have been developed to significantly improve the efficiency of Value Replacement, making the technique more useful in practice.

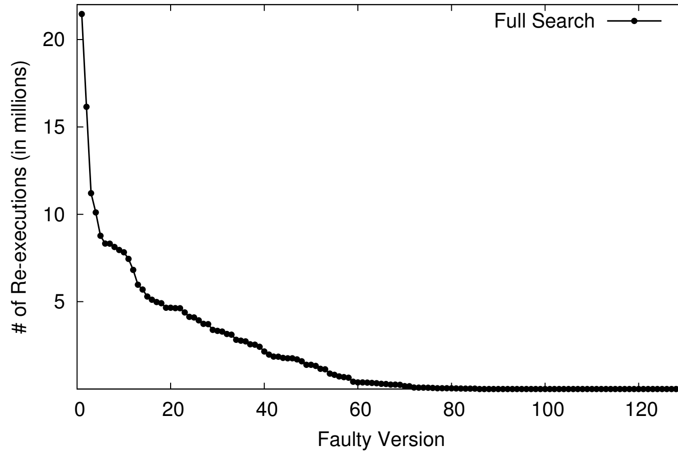

Figure 4.1: Total number of required program executions to perform value replacements using a full IVMP search, for the single-error Siemens benchmarks from Chapter 2.

4.2 Lossy Techniques to Improve Efficiency

The first set of techniques are *lossy* techniques for improving the efficiency of Value Replacement. These techniques reduce the search space for identifying IVMPs, limiting the total number of value replacements that need to be performed. It is possible with these techniques that certain IVMPs may be missed (hence the term *lossy*), since not all possible value replacements are performed. However, as the experimental results with the Siemens benchmarks have shown in Chapter 2, Value Replacement is still able to achieve highly effective error location results even with these lossy efficiency improvements in place.

The lossy techniques limit the number of statement instances that are searched for IVMPs, as well as the number of alternate value sets from the value profile that are considered when performing value replacements. These lossy techniques are now described in detail.

4.2.1 Limiting the Number of Statement Instances to Consider

To limit the number of statement instances to consider when searching for IVMPs, the following observation is made: if a statement is faulty, there is a very high chance that the statement will be associated with an IVMP (even though IVMPs can occur at other non-faulty statements too). As a result, assuming that multiple failing runs are caused by the same error, there is a very high chance that a faulty statement will be associated with IVMPs in all failing runs. This suggests the following intuition: suppose that in one failing run, IVMPs are identified in only 5 different statements; then there is a high chance that one of those 5 statements is faulty. Therefore, when considering any subsequent failing runs, the technique only needs to search at statement instances associated with those 5 statements (because any other statements are likely to be non-faulty).

Based on this intuition, the algorithm for searching for IVMPs was modified as follows. The algorithm maintains a working set of statements to consider when searching for IVMPs. This set is initialized to all statements in the first-considered failing run. After the first run is searched for IVMPs, if at least one IVMP is found, then all statements in the working set that are *not* associated with any found IVMP are removed. If no IVMPs are found, then the working set remains unchanged because the considered run does not provide any hints about which statements are likely to be faulty. The process is then repeated on the next failing run using the new working set. Under this approach, the working set can only decrease in size over time, thus limiting the number of statement instances to search in all failing runs except the first-considered run. Since all failing runs must traverse an error and the failing runs must be considered in some order, the failing runs are analyzed in increasing order of trace size. This ensures that the first-considered run is as short as

possible. A drawback of this technique is that the results may vary depending upon the order in which failing runs are considered. However, in the common case that IVMPs are indeed found at the faulty statement in all considered runs, this technique will still ensure that the ideal error location results can be achieved. Moreover, this technique can significantly reduce the search space for IVMPs in all failing runs except the first run. For instance, given a program with 500 statements, suppose that the first-considered failing run leads to IVMPs in only 2 of the statements. Then in all subsequently-considered (longer) failing runs, the statement instances associated with at most 2 different statements need to be considered.

Another technique used to limit the number of statement instances to search for IVMPs is based upon a simple observation: to rank program statements, Value Replacement only needs to know whether *at least one* IVMP is associated with a given statement in a failing run; i.e., any additional IVMPs found at a statement are useless for ranking purposes. Thus, when searching for IVMPs, the algorithm is modified so that if an IVMP is found at some statement instance in a failing run, then all other instances of the same statement in the same failing run are not searched for additional IVMPs.

4.2.2 Limiting the Number of Alternate Value Sets to Consider

When searching for IVMPs at a given statement instance, recall that all of the alternate value sets to apply when performing value replacements, are obtained from the value profile. This value profile contains, for each statement, the collection of all different value sets exercised at that statement by all test cases in an available test suite. In some cases, certain statements may be associated with many alternate value sets in the value

profile. Imagine a statement that increments a loop control variable, and suppose some test case executes the loop 1000 times; then the value profile may be associated with 1000 different value sets at this statement. Moreover, when performing value replacements, each instance of this statement in the execution would need to be used in conjunction with each of the other 999 different value sets. Thus, reducing the total number of alternate value sets to apply in value replacements can significantly speed-up the IVMP search time.

From the IVMPs studied, it was observed that in most cases where an IVMP is found at a statement, there are actually many other IVMPs that can also be found at the same statement. The intuition for this is the following. It turns out that for the Siemens benchmark programs, there are many *different* execution states that can eventually lead to the *same* output values. Often this is because of certain statements in the programs that allow for many-to-one mappings from used to defined values. One common example is a predicate statement. The final outcome of a predicate may either be `true` or `false`, but there are usually many different used values that can lead to a `true` outcome, and many different used values that can lead to a `false` outcome. Because of this, it is not surprising that any statements associated with an IVMP, are usually associated with more than one IVMP.

From the IVMPs studied, it was further observed that there is a pattern concerning the *kinds of values* that are typically associated with IVMPs. In particular, given a set of IVMPs at a statement instance with the same original value set but different alternate value sets, there is a strong likelihood that at least one of the alternate value sets in one of the IVMPs involves an alternate value that is either very close to, or very far from, the corresponding value in the original value set. For example, suppose some statement

instance uses original value 11, and assume that the value profile, for the given statement, contains 10 other alternate values that are candidates for value replacement: 2, 4, 5, 9, 12, 18, 22, 24, 34, 56. Then there is a strong chance that if IVMPs are going to be found at this statement, there will be at least one IVMP that uses value 2, 9, 12, or 56. As compared to the original value 11, these 4 particular alternate values are those that are respectively the minimum less than, the maximum less than, the minimum greater than, and the maximum greater than, the original value 11. These alternate values are referred to as the *alternate spanning values*, because they span the range of alternate values. This observation allows one to reduce the total number of alternate value sets to consider when searching for IVMPs. The Value Replacement algorithm is modified so that rather than performing value replacements using *all* alternate value sets at a given statement instance, the technique first identifies only a *subset* of alternate value sets to apply. These are the value sets that contain the alternate spanning values corresponding to the original values used at the given statement instance.

4.2.3 Value Replacement Algorithm with Lossy Efficiency Improvements

The modified Value Replacement algorithm that incorporates the lossy techniques to reduce the IVMP search space is shown in Figure 4.2. Details of this algorithm are now described in the context of the lossy techniques for reducing the IVMP search space.

Ordering failing runs. The value profile is first constructed from the provided test suite (line 1). Then, all failing runs are sorted in increasing order of trace size, where trace size is the number of statement instances in the trace (line 2). Each failing run is considered in sorted order (line 4), while maintaining a working set of all statements that

input:
Faulty program P, and test suite T containing a set F of failing runs.
output:
A ranked list of statements exercised by tests in F.
algorithm ValueReplacementRankWithLossy
begin
Step 1: [Compute IVMPs for each test in F]
1: $valProf$:= construct value profile for P with respect to T;
2: sort the test cases in F in increasing order of trace size;
3: $workingList$:= the set of statements exercised by the first failing test case in sorted F;
4: **for each** test case f in F taken in sorted order **do**
5: $trace_f$:= statement instances executed by f on P;
6: **for each** statement instance i in $trace_f$ **do**
7: s := the statement associated with instance i;
8: **if** s not in $workingList$ **then continue**;
9: $altValSet$:= alternate value sets for s in $valProf$;
10: $altValSet_{red}$:= subset of $altValSet$ with minimum/maximum values $<$ and $>$ the original values used at i;
11: **for each** alternate value set v in $altValSet_{red}$ **do**
12: **if** s has an IVMP in f **then break**;
13: **if** applying v at i corrects f's output **then**
14: **report** a found IVMP at s in f;
endfor (each alternate value set)
endfor (each statement instance)
15: **if** f has at least one IVMP **then**
16: remove stmts from $workingList$ that are not associated with any IVMP in f;
endfor (each failing run)
Step 2: [Use IVMPs to rank program statements]
17: $stmts$:= set of statements exercised by tests in F;
18: **for each** statement s in $stmts$ **do**
19: compute $suspiciousness(s)$;
20: compute $suspiciousness_{tarantula}(s)$;
endfor
21: $stmts_{ranked}$:= sort $stmts$ by $suspiciousness$, break ties by $suspiciousness_{tarantula}$;
22: **output** $stmts_{ranked}$;
end ValueReplacementRankWithLossy

Figure 4.2: The Value Replacement technique with lossy efficiency improvements.

need to be searched for IVMPs in the currently-considered failing run. The working set is initialized to all statements exercised by the first failing run (line 3). After the search for IVMPs in the current failing run completes, then if no IVMPs are found, the working set remains unchanged. If at least one IVMP is found, then any statements in the working set that are not associated with any IVMPs identified from the current failing run are removed from the working set (lines 15-16).

Limiting statement instances and alternate value sets to consider. When searching for IVMPs in a particular failing run, only the statement instances from those statements that are in the working set are considered (line 8). At each considered statement instance, the algorithm applies only those alternate value sets for which the original value of a used or defined variable at that instance would be changed to be one of the following four alternate values: (1) the minimum alternate value less than the original value; (2) the maximum alternate value less than the original value; (3) the minimum alternate value greater than the original value; and (4) the maximum alternate value greater than the original value (line 10). Additionally, whenever an IVMP is found in a failing run, then all subsequent instances of that statement in the failing run need not be searched for further IVMPs (line 12). After identifying IVMPs, the statements exercised by the failing runs are ranked (lines 17–21).

According to this algorithm, the total number of program executions required to search for IVMPs is bounded by $O(F \times I \times V)$, where F is the number of failing runs, I is the size of the *shortest* failing run execution trace, and V is the maximum number of alternate value sets to apply at any given statement. However, the technique for limiting the number of alternate value sets to consider reduces V to a small constant. Moreover, the

work in [67] suggests that relatively effective error location results can be achieved with a small F, such as only a few failing runs. As a result, the runtime of the algorithm is mostly influenced by I.

4.3 Lossless Techniques to Improve Efficiency

The second set of techniques are *lossless* techniques for improving the efficiency of Value Replacement. Rather than reducing the search space for IVMPs, these lossless techniques are instead improvements to the underlying implementation for performing value replacements, rather than modifications to the Value Replacement algorithm itself.

Under a brute-force implementation, value replacements are performed one-after-the-other, and each individual value replacement requires a complete program execution, from start to finish. A complete program execution is needed for each value replacement because only one value replacement is performed on each execution, and the resulting output of each execution needs to be observed in order to identify potential IVMPs. However, this can be very time consuming when many value replacements need to be performed.

Two observations can be made about the task of performing value replacements that allow one to significantly speed-up the computation time. The first observation is that within a failing run, performing value replacements involves a significant amount of redundant program execution. The second observation is that each value replacement can be performed in isolation, and therefore the search for IVMPs is inherently parallelizable. Based on these two observations, two corresponding implementation improvements are described that significantly speed-up the time required to perform value replacements: (1) removing redundant execution when performing value replacements; and (2) parallelizing

the task of performing value replacements. These improvements are illustrated in Figure 4.3. These two improvements are now described in detail.

4.3.1 Removing Redundant Program Execution

There is a significant amount of redundant program execution when performing value replacements in a failing run. This is because in a value replacement program execution, the part of the execution *before* the value replacement is the same as in the original failing run; execution is affected only from the point of the value replacement onwards. Many different value replacement executions in the same failing run therefore results in a significant amount of redundant execution. This is illustrated in Figure 4.3 (A). In the figure, an original program execution is shown with 3 statement instances; assume 2 value replacements are performed at each instance, for a total of 6 value replacements. Figure 4.3 (B) shows each of these 6 value replacement executions, along with the duplicated portions of the original execution.

To remove this redundant execution, a mechanism is needed that allows for the following: at a statement instance in a failing run at which a value replacement needs to be performed, the technique needs to be able to perform the value replacement and then *return directly to the same point*, without re-executing everything before this point, so that the technique can continue performing additional value replacements. In this way, all redundant execution is avoided prior to the point of a value replacement. The `fork` function in `C` provides just this functionality.

The method for removing redundant program execution is as follows. For a given failing run, the run is executed only once, from beginning to end, to perform all associated

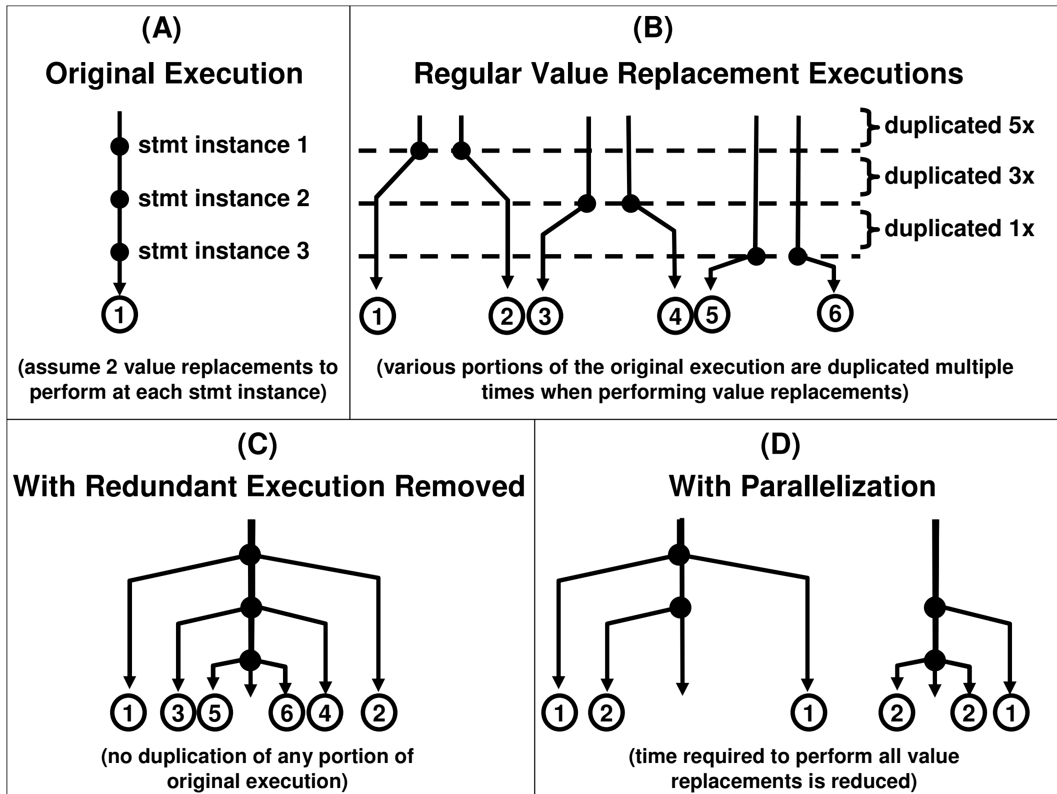

Figure 4.3: Lossless improvements to the efficiency of Value Replacement. The circled numbers indicate the relative time at which execution terminates.

value replacements. At each statement instance during execution at which one or more value replacements need to be applied, `fork` is invoked to create a child process to carry out each value replacement. When the `fork` function call occurs, a new child process is created which is identical to the parent (original) process, except for a new process ID. The parent process then waits at that statement instance as each child completes its value replacement. Afterwards, the parent process moves onto the next statement instance, where more children may be forked to perform more value replacements. This eliminates all redundant portions of execution, as depicted in Figure 4.3 (C).

In the implementation, the input and output of the parent and child processes had to be specially handled to ensure they are not affected by the forking, because `fork`

duplicates file pointers, which can cause intermixed input and output. This was handled by initially capturing all the input and output of the original parent process (representing the original failing execution), during a preliminary "input/output capturing phase." During the regular value replacement execution, when a child process is forked, then its set of input and output is specifically adjusted, such as by setting new file pointers, to match that of the parent at that point. Reads from `stdin` are handled by writing these input values to a file during the input/output capturing phase, then reading from this file as necessary during the regular value replacement execution.

4.3.2 Parallelizing the Search for IVMPs

Each time a value replacement is performed when searching for IVMPs, it is done in isolation from all other value replacements. Thus, multiple available cores can carry out multiple value replacements in parallel. To take advantage of this, the task of performing value replacements is parallelized in two ways. First, the set of all value replacements to perform are partitioned into N task sets, where N is the number of available cores. Then each task set is handed off to an available core for processing. Second, when a parent process forks C children to perform C value replacements at a statement instance during execution, then those C children can simply be allowed to execute in parallel. This parallelization is illustrated in Figure 4.3 (D). In the figure, the 6 value replacements are partitioned into 2 task sets. The figure also assumes that enough idle cores are available to process all children in parallel at each statement instance. The circled numbers in Figure 4.3 (D) show multiple value replacement executions terminating at the same relative time unit; this is due to multiple value replacements being performed in parallel.

4.4 Efficiency Results

4.4.1 Efficiency of Locating Single Errors

Siemens Benchmark Programs

In Section 2.5, a set of experiments was described in which Value Replacement was used to locate the single errors contained in the Siemens benchmark programs. These experiments actually used the modified algorithm for Value Replacement from Figure 4.2, in which the *lossy* techniques were implemented to improve efficiency by reducing the IVMP search space. To study the effectiveness of these lossy techniques in reducing the IVMP search space, a count was taken of the total number of program executions actually performed for each faulty version in order to carry out the (reduced number of) value replacements, according to the lossy algorithm in Figure 4.2. This is called the *reduced search*. These values were then compared to the total number of program executions that would have been required if a full search had been performed, without using the lossy techniques. This is called the *full search*. The comparison between the reduced search and the full search is shown in Figure 4.4.

As shown in the figure, the total number of executions actually performed using the reduced search was significantly lower than what would have been required if the technique had performed all value replacements and fully searched for all possible IVMPs. For the full search, 4 faulty versions would have required over 10 *million* program executions each to fully search for IVMPs. In fact, one of these faulty versions would have required over 20 million program executions. On the other hand, the maximum number of executions required for any faulty version using the reduced search was only about 412,000. A large majority of

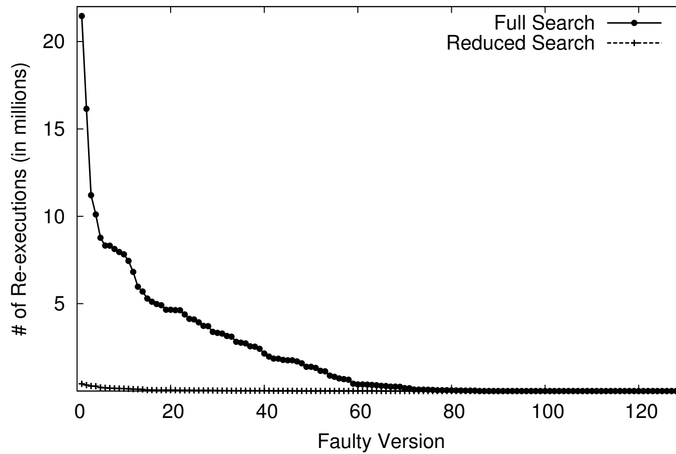

Figure 4.4: Total number of program executions to perform value replacements using the full and reduced IVMP searches, for the single-error Siemens benchmarks from Chapter 2.

faulty versions (84 out of 129) required fewer than 10,000 program re-executions to search for IVMPs using the reduced search. The same was true for only a minority of cases (44 of them) using the full search. On average across all faulty versions, the full search requires over 2 million program executions per faulty version, while the reduced search requires just under 30,000 program executions (a reduction by a factor of 67). However, note that these average values are skewed due to a few faulty versions that require unusually many program executions. In general, the few cases where the reduced search required relatively more program executions than other reduced-search cases, was due to faulty versions where the approach was not able to find any IVMPs. In these cases, all instruction instances of all failing runs were fully searched since no IVMPs were found to limit the number of statement instances to search.

There is a drastic reduction in the number of program executions required to search for IVMPs using the reduced search as compared to a full search. Figure 4.5 shows

the actual time required to search for IVMPs using the reduced search when running Value Replacement. The x-axis represents the time in minutes to search for IVMPs from all failing runs using the reduced search. The y-axis shows the percentage of faulty versions that were fully searched in less than the specified amount of time. Note that the actual time to rank statements with the computed IVMP information (and breaking ties with the Tarantula formula) is negligible compared to the IVMP search time. Computation of the value profiles for each faulty version was also very small relative to the IVMP search time, never taking more than a few dozen seconds per faulty version.

From this figure, it can be seen that most faulty versions required relatively little time to search for IVMPs using the reduced search; 50 faulty versions, many from the `tcas` program, require less than 1 minute to search all failing runs for IVMPs. A large majority of cases, 77 of them, require less than 10 minutes of search time. Almost 90% of cases, 112 of them, require less than 100 minutes. Only 17 out of the 129 faulty versions actually require more than 100 minutes of search time. The maximum required time to search was just over 14 hours (840 minutes) for one particular faulty version, but this was an unusual case where failing runs were relatively long and no IVMPs could be found to limit the number of statement instances to consider.

Larger Subject Programs

In Section 2.5.3, an experiment was conducted using a set of 5 benchmark programs that are significantly larger than the Siemens benchmarks, to show that Value Replacement can also yield effective error location results on larger programs. Table 4.1 shows the corresponding times required to search for the IVMPs, as well as the total number of executions

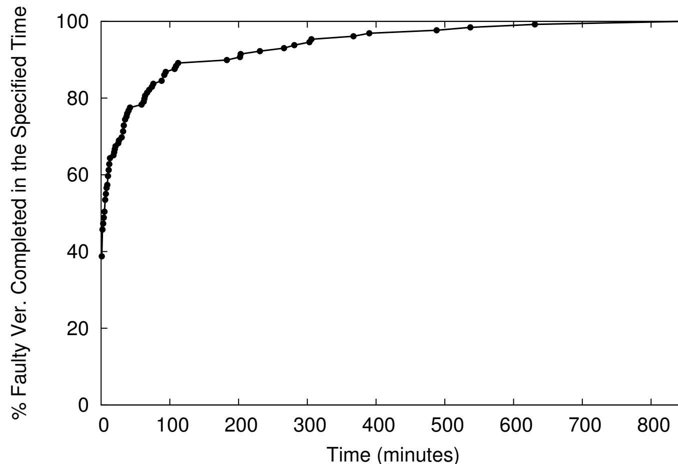

Figure 4.5: Time required to search for IVMPs in each faulty version using the reduced search, for the single-error Siemens benchmarks from Chapter 2.

performed when carrying out value replacements, as compared to the total number of executions that would have been required using a full IVMP search.

For the `grep`, `sed`, and `flex` programs, IVMP search times were quite low due to the fact that the failing runs had very short execution traces. For the `space` and `gzip` programs, IVMP search times were comparatively longer because of longer execution traces. In general, the Value Replacement technique may still require computation time on the order of hours to locate errors, even when using the lossy techniques to improve efficiency developed in this chapter. This suggests that the lossless techniques are also very important for improving the scalability of Value Replacement.

Program Name	IVMP Search Time	# IVMP Executions Done/Possible
space	79.5 min	35841/1061154 (3.4%)
grep-2.5	0.8 min	241/588 (41.0%)
sed-4.1.5	1.8 min	881/5816 (15.1%)
flex-2.5.1	0.5 min	87/228 (38.2%)
gzip-1.3	215.6 min	126845/6918816 (1.8%)

Table 4.1: Efficiency results using the reduced search for the larger benchmarks from Chapter 2.

4.4.2 Efficiency of Locating Multiple Errors

As shown in the previous section, the lossy techniques for reducing the IVMP search time can significantly reduce the total number of program executions required to search for IVMPs. However, in cases where failing execution traces are relatively long, the experiments showed that certain errors may still require time on the order of hours to locate. In order to generalize Value Replacement into an iterative technique for locating multiple simultaneous errors, it was necessary to implement and use the *lossless* efficiency improvements described in Section 4.3. These improvements drastically improved the efficiency of Value Replacement and enabled the practical use of Value Replacement in an iterative manner to locate multiple errors.

The experimental results describing the effectiveness of Value Replacement on the multiple-error Siemens benchmark programs (shown in Section 3.2) were obtained by implementing the lossless techniques to perform value replacements more efficiently. Here, the corresponding experimental results illustrating the efficiency of the technique are described. Note that the lossy techniques were not used in these experiments since the experiments completed in reasonable time without them. Moreover, one of the lossy techniques (the one that limits the number of statement instances to consider based on where IVMPs are found in previously-considered runs) implicitly assumes that all considered runs fail due to the same error; in the multiple-error Siemens programs, this assumption does not hold.

Figure 4.6 shows the total time in seconds required to search for all IVMPs when running each of the four techniques described in Section 3.2. For each benchmark program, the displayed timing data is the average from among all faulty versions associated with the program. Each bar is stacked to show the average time required for each iteration of the

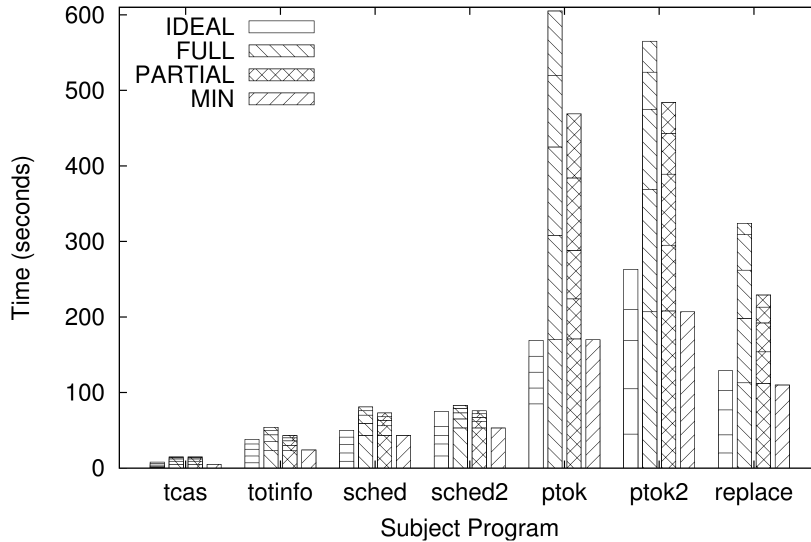

Figure 4.6: Average total time to search for IVMPs, for the multiple-error Siemens benchmarks from Chapter 3

technique (except MIN, where IVMPs are computed in only one iteration).

It can be seen in the figure that the FULL technique requires more time overall than PARTIAL, which requires more time than MIN. In general, the time required by PARTIAL is slightly closer to the time for FULL as opposed to the time for MIN. This is not surprising, considering that only FULL and PARTIAL iteratively compute IVMPs (though PARTIAL may search fewer total failing runs than FULL). However, in some cases the overall time required by PARTIAL is noticeably less than FULL. In `ptok`, FULL requires about 600 seconds (10 minutes) on average to search for IVMPs in each faulty version, whereas PARTIAL requires only about 470 seconds (under 8 minutes), a 20% reduction in running time. For `replace`, FULL requires about 5.5 minutes while PARTIAL requires about 3.8 minutes, a 30% reduction in running time. Since the effectiveness of PARTIAL is similar to that of FULL (especially for `ptok2` and `replace`), then PARTIAL may be useful in situations where running time is an issue. Note also that the timing results for

IDEAL are generally less than for FULL and PARTIAL; this is due to the programs being single-error versions in IDEAL, with relatively fewer failing runs exposing these errors that need to be searched for IVMPs.

Another observation to make about the data in Figure 4.6 is that all three multiple-error techniques (i.e., FULL, PARTIAL, and MIN) require about the same amount of time to search for IVMPs for the very first iteration. This is because all three techniques compute IVMPs for all failing runs in the suite on the first iteration. The PARTIAL technique may then possibly refrain from searching some failing runs in subsequent iterations. The MIN technique will never re-compute IVMPs after the first iteration.

Table 4.2 shows the actual number of failing runs that are searched for IVMPs in each iteration, on average for each faulty version in a benchmark program. For example, for the FULL technique in program `replace`, the first iteration required a search for IVMPs in an average of 9 failing runs, the second iteration required a search in an average of 7 failing runs, and so forth, such that an average of 26 runs needed to be searched in total.

The results in this table mirror the timing results from Figure 4.6; PARTIAL requires IVMP searches in fewer total runs than FULL. MIN requires IVMP searches in the fewest number of runs. An interesting observation to make from this table is that as the iteration number increases for both the FULL and PARTIAL techniques, the total number of failing runs that must be searched for IVMPs tends to decrease. This is because as errors are iteratively found and fixed over time, fewer failing runs will tend to remain.

Program Name	**Average Number of Runs to Search for IVMPs**			
	IDEAL	**FULL**	**PARTIAL**	**MIN**
tcas	1+1+2+2+2=8	5+4+3+2+1=15	5+4+3+2+1=15	5
totinfo	3+4+4+3+3=17	10+5+4+3+2=24	10+3+3+2+1=19	10
sched	2+3+3+2+2=12	10+4+3+2+1=20	10+3+2+1+1=17	10
sched2	5+5+3+4+6=23	9+4+3+2+1=19	9+3+2+2+1=17	9
ptok	4+1+1+1+1=8	8+7+6+5+4=30	8+2+3+5+4=22	8
ptok2	1+1+2+1+1=6	5+4+3+1+1=14	5+2+2+1+1=11	5
replace	2+2+3+2+2=11	9+7+5+4+1=26	9+3+3+2+1=18	9

Table 4.2: Average number of runs searched for IVMPs (separated by iteration number).

4.5 Summary

In this chapter, efficiency issues related to Value Replacement were discussed. A set of *lossy* and *lossless* techniques were developed that can significantly improve the efficiency of Value Replacement. The lossy techniques improve efficiency by significantly reducing the search space for IVMPs. This is accomplished by limiting the number of statement instances and alternate value sets to consider when searching for IVMPs. On average, this reduces the total number of value replacements that need to be performed in our benchmarks from Chapter 2 by a factor of 67. These efficiency improvements come at the expense of a possible loss in precision, since some IVMPs may be missed. However, the experimental results from Chapter 2 indicate that highly effective error location results can still be achieved when the lossy efficiency improvements are used. The lossless implementation improvements drastically improve the efficiency of performing value replacements, without actually reducing the IVMP search space. This is accomplished by eliminating redundant execution when searching for IVMPs, and by parallelizing the IVMP search. On average, these implementation improvements allow multiple simultaneous errors from our benchmark programs in Chapter 3 to be located in minutes in the worst case.

The Value Replacement technique, as implemented, does not handle address values and therefore may have limited effectiveness for locating memory errors. Although Value Replacement can be extended to consider address values, there is a much more efficient state alteration technique that can take advantage of the unique traits of memory errors to provide more targeted state alteration, to locate memory errors more quickly. This technique, called *Execution Suppression*, is developed in the next chapter.

Chapter 5

Locating Memory Errors using Execution Suppression

In this chapter, an automated, dynamic state alteration technique called *Execution Suppression* is developed, which iteratively isolates memory corruption during program execution to locate memory errors. Isolating memory corruption is important because significant propagation of corrupt memory values can occur during execution as a result of a memory error [68, 71]; this memory corruption propagation can separate the program failure from the actual error that is the root cause of the failure, concealing these errors and making the task of locating them challenging.

Suppression is used to identify the first point of memory corruption in an execution that fails due to a crash caused by a memory error. It is assumed that the first point of memory corruption is either at, or very close to, the memory error; as will be shown in a study of memory errors and memory corruption described in Section 5.1, this assumption is reasonable. Suppression is the idea of *omitting one or more instructions during program*

execution. The concept of suppression is iteratively used to gradually isolate the first point of memory corruption. When a crash occurs, this reveals that the memory location(s) accessed at the point of the crash is corrupt. In essence, each crash reveals a subset of the memory corruption present in an execution. This subset of known memory corruption, and everything else in the execution directly or indirectly dependent upon it, is then suppressed during re-execution of the program by simply omitting the effect of the associated instructions during execution. This effectively causes only the subset of the original execution to be re-executed, that does not involve or depend upon the identified memory corruption. Note that this guarantees that the original crash, and any crashes that might have occurred due to the suppressed instructions, will be avoided in the re-execution. This is because the effect of all instructions directly or indirectly dependent upon any suppressed instruction will be omitted as well during re-execution. At the end of the re-execution, if no other crashes occur, then the last suppressed point of memory corruption is likely to be the first point of memory corruption in the execution. On the other hand, if another crash does occur, then this reveals that additional memory corruption remains in the execution, and so the process should be repeated to isolate the first point of memory corruption.

Definition 7 (Suppression).

Given a program execution, **suppression** *of one or more statement instances involves omitting the effects of these statement instances during the execution. To ensure that the execution does not become unstable due to the omitted statement instances, then any statement instances directly or indirectly dependent upon a suppressed statement instance must also be suppressed during execution.*

The idea of suppression fundamentally relies on the assumption that if memory

corruption exists in an execution, then a program crash will occur. However, this assumption does not always hold in practice because memory corruption will not always lead to a crash. Therefore, this chapter develops *variable re-ordering*, a technique that can sometimes expose crashes due to memory corruption in an execution that does not otherwise result in a crash. The intuition is as follows: even though the relative ordering of variables in memory should not affect the correctness of a program, in the presence of memory errors, the relative ordering can in fact affect where and when crashes might occur. Variable re-ordering systematically tries different variable orderings in memory to attempt to expose crashes due to memory corruption. By combining the ideas of suppression and variable re-ordering, the effectiveness and applicability of the Execution Suppression technique is greatly improved.

Definition 8 (Variable Re-ordering).

Given a program execution, **variable re-ordering** *is the process of altering the layout of global, local (stack), or heap variables in memory during that execution, such that the relative ordering of variables is different. This is performed such that program correctness is not affected, but different crashes may be exposed in the presence of a memory error.*

Execution Suppression is designed to be iterative so that the first point of memory corruption can be identified even in executions with significant propagation of memory corruption. The technique is also fully automated, which makes it useful for quickly determining whether different inputs exhibiting different program failures are due to the same error. Moreover, the technique is general and can be applied to any memory errors that involve corrupted memory and can result in a program crash.

In the next section, a study of memory errors and memory corruption is described that was conducted to motivate the development of the Execution Suppression technique.

5.1 A Study of Memory Errors and Memory Corruption

5.1.1 Memory Errors

Memory errors represent an important class of software errors that cause mishandling of memory during program execution. One example of mishandling of memory occurs when a program attempts to read from or write to an incorrect memory location. Such memory errors often manifest themselves in the form of a program crash. Examples of memory errors include the following.

- **Buffer overflows** occur when memory locations are accessed that are outside of proper buffer boundaries. Such overflows can cause unexpected corruption of program data that can eventually cause a crash. *Stack smashing* is one type of problem that can arise due to a buffer overflow, which corrupts the return address of a function on the call stack.

- **Uninitialized reads** occur when the value contained in a memory location is loaded before any proper value has been stored into that location. This can lead to unexpected program behavior due to an arbitrary value being loaded. *NULL dereferences* are a common type of uninitialized read in which the pointer used to access a memory location is unexpectedly NULL.

- **Dangling pointers** point to invalid objects in memory. One cause for this is when a memory object is explicitly deallocated, while a pointer to that object retains its

original address value. Subsequent uses of this dangling pointer can lead to unexpected program behavior.

- **Double frees** occur when a call to function `free` is performed using a deallocated address that has already been previously freed. Such an error will lead to a program abort.

In general, a memory error manifests itself by undergoing three specific events at one or more execution points during program execution.

1. **Traversal of the error**. This is when the portion of code ultimately responsible for a program failure is executed.
2. **First point of memory corruption**. Once a memory error is traversed, a first point of memory corruption may then occur, at which point memory is mishandled in some way. This can in turn cause memory to be mishandled at subsequent execution points as well, i.e., memory corruption may *propagate* during execution.
3. **Failure**. This is the point at which a developer can actually observe that a problem has occurred during execution, and realize that an error exists in the program. One type of failure that often results from memory corruption is a program crash, though not all memory corruption may lead to a crash, and there are other types of failures such as incorrect output.

Once a failure occurs during execution, a developer must find the location of the error so that the error can be eliminated. However, in general the error may not be at the same point at which the failure occurs. The portion of execution separating the traversal of

the error from the actual failure may be very long, and may involve significant propagation of memory corruption.

5.1.2 Results of the Study

A study was conducted involving 11 real programs containing known memory errors, to study how memory corruption can propagate during execution. The programs used in the study were obtained from the work of previous researchers on *BugBench*, *BugNet*, and *AccMon* [90, 100, 155], and are described in Table 5.1. The first column in the table shows the program name and version number. The second column shows the number of lines of code in thousands, measured using the `SLOCCount` tool [59]. In the third column, the following abbreviations are used to indicate the memory error type: global buffer overflow (GO); heap buffer overflow (HO); stack buffer overflow (SO); NULL dereference (ND); and double free (DF). The fourth column shows the file name and line number of the location of the memory error. The right-most column gives a brief description of the program. These subject programs were selected because they have been used as benchmarks in prior research and they contain many different types of memory errors.

The goal of the study was to understand the nature of memory errors to motivate the development of an effective technique for automatically locating them. For each memory error, the following were identified: the *location of the error*, which was known beforehand; the *first point of memory corruption*; and the *failure point*, which was always the point of execution termination, being either a crash or an observed wrong output. To specify the first point of memory corruption in an execution, the following definition for memory corruption was used.

Program Name	# Lines of Code	Error Type	Error Location	Program Description
gzip-1.2.4	6.3 K	GO	gzip.c: 828	file compression
man-1.5h1	10.8 K	GO	man.c: 979	display manual pages
bc-1.06	10.7 K	HO	storage.c: 176	arbitrary precision calculator
pine-4.44	211.9 K	HO	bldaddr.c: 7270	Internet news and e-mail
mutt-1.4.2.1	65.9 K	HO	utf7.c: 152	e-mail client
ncompress-4.2.4	1.4 K	SO	compress42.c: 886	file compression
polymorph-0.4.0	1.1 K	SO	polymorph.c: 191	filename converter
xv-3.10a	69.2 K	SO	xvbmp.c: 165	image manipulation
tar-1.13.25	28.4 K	ND	incremen.c: 180	archiving utility
tidy-34132	35.9 K	ND	parser.c: 854	HTML quality enhancer
cvs-1.11.4	104.1 K	DF	near server.c: 992	versioning system

Table 5.1: Memory error programs analyzed in the memory corruption study.

Definition 9 (Memory Corruption).

During the execution of a program, **memory corruption** *occurs when either an incorrect memory location is accessed (read or written when it should not have been), or an incorrect memory address value is assigned to a (pointer) variable.*

This definition for memory corruption captures the act of mishandling memory addresses as well as the propagation of corrupt memory address values. Note that memory corruption can propagate through other non-address values that may become infected due to memory corruption in an execution (an *infected* value is one that differs from the expected value). However, infected non-address values are not considered to be "memory corruption" according to the definition, since they cannot directly cause a crash unless they are later used to compute an incorrect memory address. Essentially, "memory corruption" is defined to be a mishandling of memory that can directly cause a program crash.

The data collected for each subject program in the study is reported in Table 5.2. For each program, a variety of different inputs were created that traversed the error and

Program Name	Input Type	Distance: [ERR→CORR] + [CORR→END]	
		# Static Dep. Edges	# Executed Instr. Inst.
gzip-1.2.4	No Crash	0 + 1	0 + 41,168
	Crash Point 1	0 + 1	0 + 36,902
man-1.5h1	Crash Point 1	1 + 8	296 + 14,239,521
bc-1.06	No Crash	1 + 0	8 + 33,567
	Crash Point 1	1 + 1	8 + 5,004
pine-4.44	No Crash	5 + 20	1,103 + 1,390,483
	Crash Point 1	5 + 14	1,103 + 10,165
mutt-1.4.2.1	No Crash	0 + 9	0 + 140,750
	Crash Point 1	0 + 8	0 + 5,697
ncompress-4.2.4	No Crash	0 + 1	0 + 7,318
	Crash Point 1	0 + 2	0 + 11,616
	Crash Point 2	0 + 1	0 + 19,637
polymorph-0.4.0	No Crash	1 + 2	4,294 + 99,723
	Crash Point 1	1 + 2	4,321 + 99,762
	Crash Point 2	1 + 1	4,354 + 113,083
xv-3.10a	No Crash	1 + 2	122 + 185,818
	Crash Point 1	1 + 1	124 + 158,640
tar-1.13.25	Crash Point 1	0 + 1	0 + 210,505
tidy-34132	Crash Point 1	0 + 2	0 + 57
cvs-1.11.4	Crash Point 1	1 + 0	5,164 + 0

Table 5.2: Study results for each analyzed input for each memory error subject program. In the "Distance" header, the following abbreviations are used: point of error traversal (ERR), first point of memory corruption (CORR), and point of execution termination (END).

triggered memory corruption. It was then observed whether or not a crash occurred and at which program statement. For each distinct execution outcome (either no crash, or a crash at a particular statement), one representative input associated with that outcome (listed in column 2) was selected and studied in detail. From the execution of each input, the *distance* from the traversal of the error until the first point of memory corruption was measured, as was the distance from the first point of memory corruption until execution termination. This distance was measured first in terms of the maximum number of *static dependence edges* (column 3), showing the extent to which memory corruption can propagate during execution. The static dependence edges were identified by manually looking at the program

code, following the chain of data and control dependencies observed in the source code. The distance was also measured in terms of the *dynamic instruction instances in the execution* (column 4), indicating the length of program execution between these execution points.

For example, for program `gzip` that has a global buffer overflow in a call to `strcpy`, one memory-corruption-inducing input was created that did not crash, and another input was created that caused a crash at one program point. For both inputs, the static dependence and dynamic instruction instance distances from the error traversal to the first point of memory corruption is 0. This indicates that in this program, the traversal of the error occurs precisely at the first point of memory corruption. On the other hand, both inputs have a static dependence distance of 1 from the first point of memory corruption until the point of execution termination, and a corresponding dynamic instruction instance distance of 41,168 instructions (the non-crashing input) and 36,902 instructions (the crashing input).

5.1.3 Key Observations

From the results presented in Table 5.2, it can be seen that static dependence distances from the point of error traversal until the point of execution termination are usually more than 1, and sometimes considerably more than 1 (e.g., programs `man`, `pine`, and `mutt`). Even when static dependence distances are relatively small, the instruction instance distances can be quite large (e.g., programs `polymorph` and `xv`). Thus, the first observation that can be made from the study concerns these total distances.

Observation 1: (Total distances). The total distance, both in terms of static dependence edges as well as dynamically-executed instruction instances, between the point of error traversal and the point of execution termination, can be large.

This observation suggests that in crashing executions, the memory error may be difficult to manually locate from the point of the crash. Traversal of the error may have occurred much earlier in time than the point of the crash. There may also be significant memory corruption propagation during execution. Thus, an automated technique to isolate the first point of memory corruption can greatly help in locating memory errors.

An interesting result pertaining to static dependence distance occurs for the non-crashing input of program `bc`. In this case, the dependence distance from the first point of memory corruption until execution termination is 0, but this is because there happens to be no memory corruption propagation during this execution. In the execution, a buffer overflow causes an unexpected write to another memory location, but this defined memory location is associated with a variable that is never accessed during the rest of the execution.

The second important observation that can be made from the results of the study deals with the types of inputs that were analyzed. All analyzed inputs triggered memory corruption, but they often had different execution outcomes.

Observation 2: (Inputs triggering memory corruption). Different inputs triggering memory corruption may lead to crashes at different program locations, or they may result in no crash at all.

If different inputs lead to crashes at different program locations, this can be misleading and may cause a developer to suspect that the inputs reveal multiple distinct errors when in fact all crashes may be due to the same error. An automated technique to help locate memory errors can help a developer to quickly group crashing inputs according to their associated errors. One useful application of this capability would be to allow developers to prioritize the fixing of errors by determining which errors are associated with the

most undesirable crashing inputs.

Inputs that trigger memory corruption but do not result in any crash may conceal the fact that a memory error exists. Even if wrong output is produced, a developer may not be able to easily tell whether the wrong output is due to a memory error or to a non-memory error. In order to improve software reliability, it would be desirable for an input that triggers memory corruption to result in a crash. This guarantees that the memory error will be revealed and encourages the developer to address the problem.

The inputs created for program `man` represent an interesting case from among the programs with buffer overflows. This is the only program with a buffer overflow in which inputs could only be created that trigger memory corruption and then subsequently crash. For this program, it turns out that the error is such that if at least one memory location gets corrupted, then a crash will happen. Thus, no inputs could be created for this program that triggered corruption but did not crash.

A third observation that can be made concerns the relative distance from the point of error traversal to the first point of memory corruption, compared to the distance from the first point of memory corruption to the point of execution termination.

Observation 3: (Relative distances). Across all programs in the study, the relative distance from the point of error traversal to the first point of memory corruption, is generally *considerably less* than the distance from the first point of memory corruption to the point of execution termination.

In all programs except `pine`, the maximum static dependence distance from the point of error traversal to the first point of memory corruption is always 0 or 1. As a result, an automated technique to isolate memory errors can still be very effective if it seeks to only

isolate the first point of memory corruption. From the first point of memory corruption, a developer should only need to exert minimal effort to find the actual error.

Together, the three observations resulting from the study of memory errors and memory corruption motivate the Execution Suppression technique that is based on the ideas of suppression and variable re-ordering. The next section describes in detail this technique for isolating the first point of memory corruption in an execution that fails due to a memory error.

5.2 Isolating the First Point of Memory Corruption using Suppression

Suppression involves *omitting the effect of one or more statements during program execution.* This idea can be used iteratively to reveal the first point of memory corruption in an execution. In general, when there are multiple instances of memory corruption that exist in an execution, then eventually one of these instances may cause a crash. If one suppresses (i.e., omits execution of) the associated statements directly causing this crash, as well as any other statements directly or indirectly dependent upon these suppressed statements, this would avoid the crash and allow execution to proceed further. This provides opportunity for the remaining memory corruption to cause other crashes in the execution. This, in turn, reveals more of the memory corruption, until finally the first point of memory corruption is revealed. The first point of corruption is assumed to be identified when no further crashes occur, because suppressing the first point of memory corruption will ensure that the program will not produce any further crashes. This is the essence of suppression.

5.2.1 Motivational Example

To illustrate the functionality and usefulness of suppression, consider the sample code presented in Figure 5.1. In this snippet of code, there exists a copy-paste error at line 4; the programmer copies lines 1 and 2, pastes into lines 3 and 4, and then forgets to change variable x into variable y at line 4. The effect of this error is that pointers $p2$ and $q2$ mistakenly refer to the same memory location. As a result, when a value is stored into location $*q2$ at line 8, then this clobbers the value originally stored there at line 6. Any subsequent uses of the value at location $*p2/*q2$ then make use of an infected memory location, which can lead to further infection at other memory locations (e.g., at lines 9, 10, and 11). Essentially, the error at line 4 immediately causes memory corruption that propagates through multiple locations until eventually a program crash may occur (potentially at lines 12, 13, and 15).

Suppose the code in Figure 5.1 is exercised on some input. This is represented pictorially in Figure 5.2 (A). Initially, pointer $q2$ is corrupted at line 4 since it points to an incorrect memory location. That memory location is then infected at line 8, where the value previously stored at that location is mistakenly overwritten. Then, the definition at location a (line 9) is infected since it uses the infected value from location $*p2/*q2$. The definition for location b (line 10) is similarly infected. This further results in infection of location c (line 11). Now, suppose that the program crashes at line 12 due to infected array index c accessing an illegal address outside the boundary of array $intArray$. This is a buffer overflow failure. When the failure is observed at line 12, identifying the root cause at line 4 is not obvious since in practice one does not know the first point of memory corruption and how the corruption might propagate during execution.

Let x and y be pointers to two malloc'ed memory regions, each able to hold two integers.
Let $intArray$ be a heap array of integers.
Let $structArray$ be a heap array of pointers to structs with a field f.

```
1:  int * p1 = &x[1];
2:  int * p2 = &x[0];
3:  int * q1 = &y[1];
4:  int * q2 = &x[0]; // copy-paste error: should be &y[0]
5:  *p1 = readInt();
6:  *p2 = readInt();   // gets clobbered at line 8
7:  *q1 = readInt();
8:  *q2 = readInt();   // clobbers line 6 definition
9:  int a = *p1 + *p2;  // uses infected *p2/*q2
10: int b = *q1 + *q2;  // uses infected *p2/*q2
11: int c = a + b + 1;  // uses infected a and b
12: intArray[c] = 0;    // potential buffer overflow
13: structArray[*p2]->f = 0;  // potential NULL dereference
14: free(p2);
15: free(q2);  // potential double free
```

Figure 5.1: Example to illustrate the functionality and usefulness of suppression.

As a first step to begin searching for the root cause of the program crash at line 12, the program can be re-executed while suppressing the memory corruption currently known that directly causes the crash. This is depicted in Figure 5.2 (B). To do this, notice at line 12 that the value at location c is used, along with the base address for variable $intArray$, to compute the effective memory address to access. Since either of these used locations could be infected, one should identify the last definitions of both (for $intArray$, this is not shown in the figure). Location c is last defined at line 11. The program is the re-executed on the same input, but during execution, suppression is performed on the definition of the base address for $intArray$ (not shown in the figure) as well as the definition of c at line 11 (by *not* performing the store to location c). Accordingly, execution of any subsequent statements

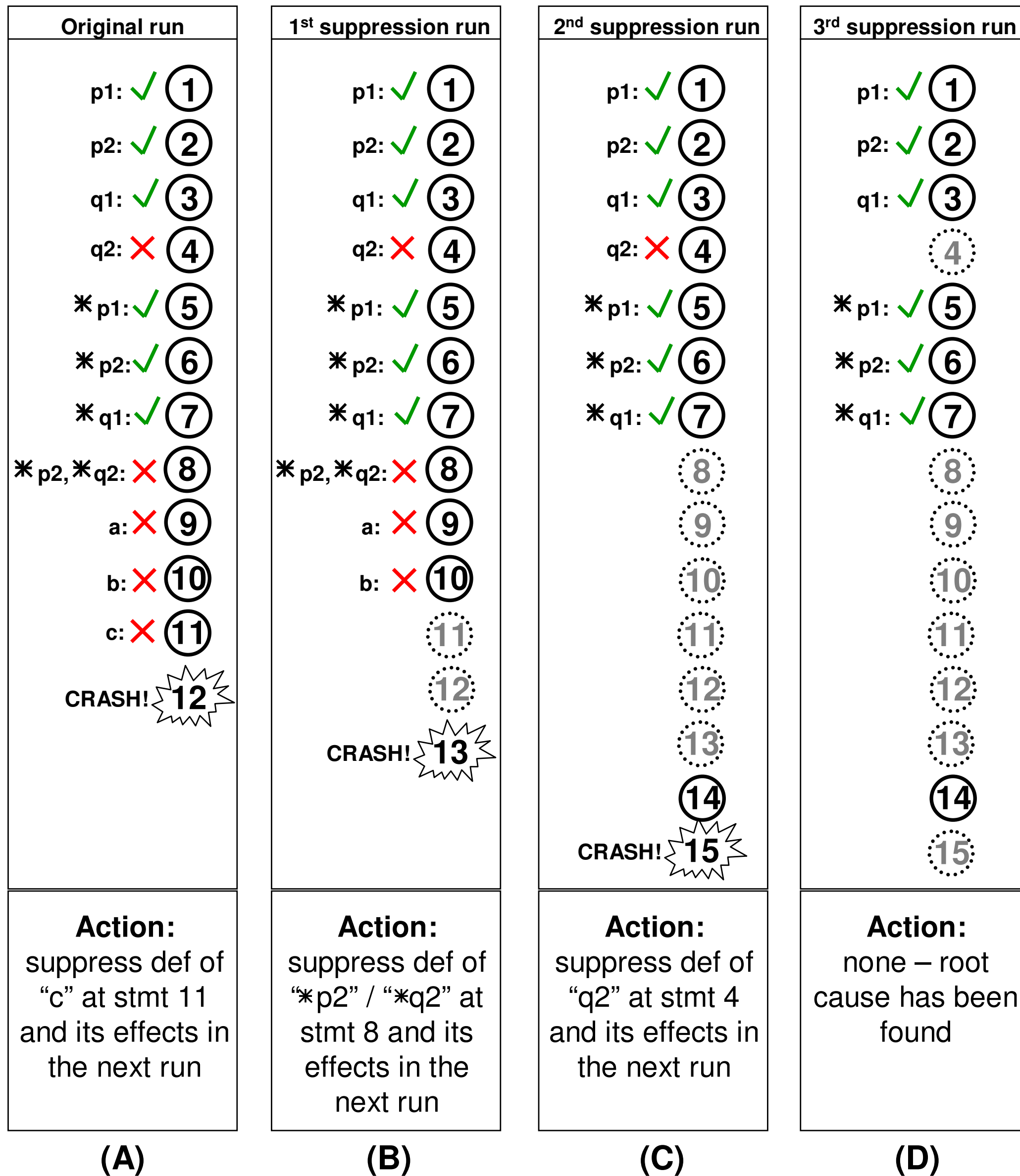

Figure 5.2: Suppression executions for the example code in Figure 5.1. Solid circles are executed statements, and dotted circles are suppressed statements. Statements defining a memory location are annotated with information showing whether the location is infected (x) or not infected (check).

directly or indirectly influenced by these definitions are also suppressed. In the example, only lines 11 and 12 are suppressed when the program is re-executed. However, suppose that now the execution reaches line 13 and another crash occurs. This is possible since infected location $*p2$ is used as an index into an array of struct pointers. In the example, suppose that $structArray[*p2]$ is actually NULL. Then line 13 will result in a segmentation fault since NULL is dereferenced. The root cause of this failure is still at line 4, but its location is not obvious at this point.

The program is re-executed again to suppress the newly-revealed memory corruption directly involved in the crash at line 13. This is depicted in Figure 5.2 (C). This time, suppression is performed on the last definition of location $*p2$, which occurs at line 8, plus the other statements that are directly or indirectly influenced by the definition at line 8 (similarly for the last definition of the base address for $structArray$, not shown in the figure). Note that line 8 is the appropriate last definition of $*p2$, since pointer $q2$ actually refers to the same location as $p2$. In the example, during execution, suppression is performed on lines 8, 9, 10, 11, 12, and 13. Note that lines 14 and 15 are not suppressed since the infected location defined at line 8 (which happens to be pointed to by both $p2$ and $q2$) does not actually influence the locations of the pointers $p2$ and $q2$ themselves used at lines 14 and 15. With these new suppressions, however, the program crashes yet again. At line 15, the program aborts due to a double free of the same memory location (last defined at line 4).

Finally, the program is re-executed a third time as shown in Figure 5.2 (D). Here, suppression is performed on the definition of pointer $q2$ at line 4, plus its subsequent use at line 15. In total, only lines 1, 2, 3, 5, 6, 7, and 14, are executed. In this case, the program

proceeds normally (without any crash) since all memory corruption has been suppressed during execution. In other words, only the statements involving un-infected memory locations are exercised. As a result, it can be concluded that the most-recently identified statement for suppression – line 4 – is directly associated with the memory error, because it is the root cause of all the memory corruption that led to the program crashes. Overall, this example shows how suppression gradually isolates the first point of memory corruption by suppressing program crashes to iteratively reveal more memory corruption. This continues until the first point of memory corruption is revealed and suppressed, resulting in no additional program crashes.

5.2.2 The Suppression Algorithm

The algorithm to carry out suppression is shown in Figure 5.3. The technique requires as input a program and an associated test case for which a program crash occurs due to memory corruption. The technique iteratively searches for points of memory corruption in the execution and suppresses the effects of these points until the first point of memory corruption is found (which is assumed to be at or near to the memory error). The technique iterates as long as a program crash occurs. It is assumed that if memory corruption exists in an execution, then a crash will occur. Thus, when no further crashes occur, the most recent point(s) of suppression is assumed to be the first point of memory corruption in the execution.

The main loop comprising the technique is shown in lines 3 – 10 in Figure 5.3. This loop iterates as long as a crash occurs. On each iteration, the corrupted/infected memory location and the associated statement instance causing the crash are identified (lines 4 and

```
input:
    Program P and test case t causing a crash due to a memory bug.
output:
    Stmt(s) identified as the first point of memory corruption in execution of t on P.
algorithm SuppressionTechnique
begin
1:  S_def := "undefined";
2:  Initial_Suppression_Points := {};
3:  while a program crash C occurs during execution of t on P do
4:     S_crash := the statement instance directly causing crash C;
5:     Loc := accessed memory location(s) causing crash C at stmt instance S_crash;
6:     S_def := the statement instance(s) originally defining the value(s) in Loc
          prior to its use at S_crash;
7:     if S_def does not exist then
8         S_def := the statement instance(s) originally defining the address(es) of Loc
             prior to its use at S_crash;
       endif
9:     Initial_Suppression_Points := Initial_Suppression_Points ∪ {S_def};
10:    re-execute t on P while suppressing (nullifying) the effects of
          (1) all statement instances in Initial_Suppression_Points;
          (2) all statement instances directly/indirectly influenced by some statement
             instance in Initial_Suppression_Points
    endwhile
11: report the program statement(s) associated with the latest S_def;
end SuppressionTechnique
```

Figure 5.3: The suppression algorithm to identify the first point of memory corruption in an execution that crashes due to a memory error.

5). In some cases, such as crashing array accesses that involve both a base address as well as an index value, there may be more than one location that could be infected; all such locations are considered. The statement instance that defined this corrupted memory location is identified (line 6). However, an accessed memory location may have no prior definition in cases where the address of the accessed location itself is incorrect. In this case, the statement instance that defined the incorrect address is identified instead (lines 7 and 8). In the case where more than one location is identified at line 5, the same number of associated statement instances will be identified in line 6 or 8. Lines 6 through 8 essentially

give preference to the possibility of an incorrect value in a memory location as opposed to the possibility of the memory location itself being incorrect. However, this approach is effective and worked well in the experiments. This is because if the memory location itself is incorrect, then it is unlikely for there to be a prior definition to that location, and so line 8 is highly likely to be executed in this case.

When identifying the last definition of an infected value, the *original* definition of the infected value is identified (bypassing any copies that may occur, for instance, by passing values through function calls). This is because the original definition of an infected value is the one that is ultimately responsible for the crash caused by that infected value. Once the definition statement(s) is identified, it is then added to the set of "initial suppression points," the execution points at which suppression should be initiated upon program re-execution (line 9). The key step of the technique is then performed (line 10), in which the program is re-executed using the same input. During execution, the direct and indirect effects of all statement instances in the set of initial suppression points are suppressed. The effect of this suppression is that the previously-occurring crash will be avoided, since the memory corruption directly causing it will have been suppressed. Thus, either a new crash will occur in the execution – in which case the loop iterates again – or else no crash will occur and the last-identified suppression point is identified as the likely first point of memory corruption in the execution (line 11). On rare occasions, more than one likely first point of memory corruption may be outputted by the technique at line 11, since lines 6 and 8 may identify more than one statement instance. In these situations, a developer may have to manually analyze a few statements in order to identify the true first point of corruption. However, this situation did not arise in the experimental study described later in this chapter.

5.2.3 Real-world Example using the Suppression Algorithm

The use of suppression on the real-world example depicted in Figure 5.4 is now illustrated. This figure shows a crashing execution associated with a memory error in the `pine` program, which is a program for Internet news and e-mail. In the figure, the root cause (error), the first point of memory corruption, and the point of the crash, are highlighted. Solid arrows between statements represent propagation of incorrect values during the crashing execution: thin arrows from the point of the root cause until the first point of memory corruption represent propagation of non-address incorrect values; thick arrows from the first point of memory corruption until the crash show propagation of memory corruption. Arrows with dotted lines represent control flow. To follow the path leading up to the failure, start at the point of the root cause and follow the arrows until the point of the crash is reached.

In this example execution, the memory error that is root cause of the failure occurs in file `bldaddr.c`, line 7270. At this statement, a size value is estimated to be too small because it does not account for the possibility of special characters in an input string. This infected size value propagates through several statements until it is used at line 7126 to allocate a heap buffer. This pointer variable assignment is considered to be the first point of memory corruption, because the buffer is allocated based on an incorrect size. A pointer to this buffer is then passed through several function calls until function `rfc822_cat` in file `rfc822.c` is executed. Within this function, data is written into the buffer and the buffer is overflowed. Control then eventually reaches file `bldaddr.c`, line 7134, where the pointer to the infected buffer is returned to file `mailindx.c`, line 4502. The pointer is finally passed to file `fs_unix.c`, line 60, where a call to `free` occurs that finally results in a program crash.

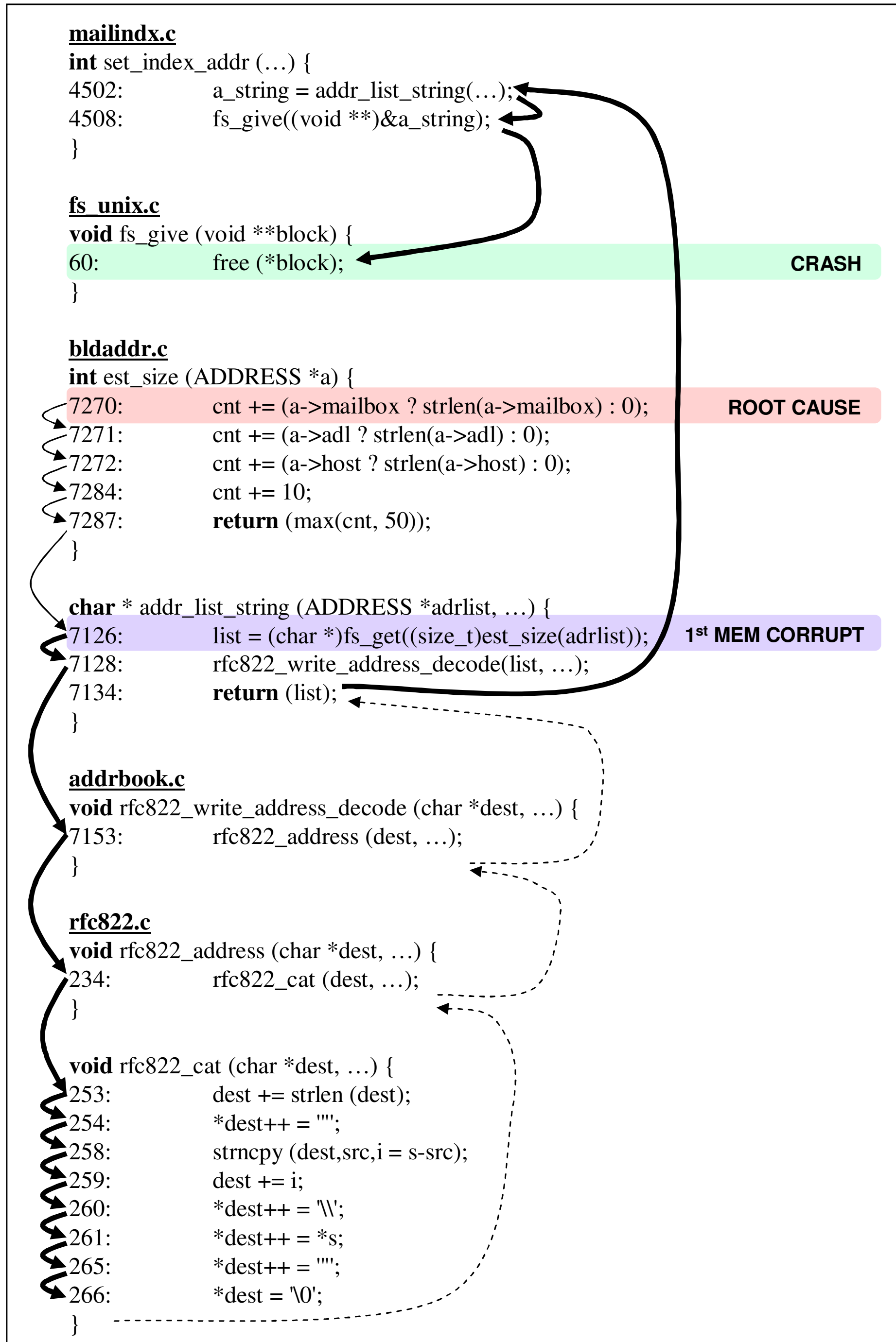

Figure 5.4: A failing execution in the pine program illustrating how traversal of a memory error can lead to memory corruption and ultimately trigger a crash. Thin solid arrows between statements represent propagation of incorrect non-address values. Thick solid arrows represent memory corruption propagation. Dotted arrows represent flow of control.

The crash occurs because of the earlier buffer overflow corrupting an important value that is needed by the `free` function.

It turns out that for this execution, there are 5 static dependence edges (1,103 dynamic instruction instances) from the point of error traversal until the first point of memory corruption, and an additional 14 static dependence edges (10,165 dynamic instruction instances) from the first point of memory corruption until the point of the crash. Further, 19 static dependence edges and over 11,000 dynamic instruction instances separate the root cause from the crash. Thus, there is considerable propagation of incorrect values and corrupt memory locations in this crashing execution. This illustrates the potential for memory errors to have complicated effects on program execution, and demonstrates why memory errors can be difficult to locate.

When running the suppression technique on this program execution to isolate the first point of memory corruption, it is determined that the program crash in function `free` is caused by an unexpected software abort, due to accessing a single particular memory location. The original definition of this location happens to be the definition of variable *list* at line 7126 of file `bldaddr.c`. This is actually the first point of memory corruption, since the buffer allocated at this point is too small. As a result, the suppression technique re-executes the program while suppressing this definition and all of its effects. In this case, all memory corruption is avoided (since all statements influenced by the too-small buffer are suppressed) and therefore no program crash occurs. The technique then reports the correct first point of memory corruption in the execution. Thus, even though `pine` has a large degree of memory corruption propagation in this case, the technique is able to bypass many memory corruption dependence edges when isolating the first point of memory corruption.

Whether or not this can happen in other situations depends upon which infected memory locations directly cause a crash in an execution. In the case of `pine`, it happened that the first corrupted memory location directly led to the first crash, so the technique identified the ideal result very quickly. However, in general the suppression technique may sometimes require several iterations to identify the first point of memory corruption.

5.3 Exposing Program Crashes using Variable Re-ordering

The suppression technique relies on the fundamental assumption that memory corruption in an execution will cause a program crash. This assumption may not hold in cases where corrupted memory is never accessed, or in cases where it may be accessed in such a way that no crash happens to occur. Evidence that this can happen was seen in the memory corruption study (Section 5.1), in which inputs could be created in certain cases that trigger memory corruption but do not result in a crash. As a result, this exposes two limitations of suppression. First, the technique is not applicable to executions that do not originally crash, even though the executions may traverse an error and cause memory corruption. Second, the technique may terminate prematurely in cases where no crash occurs during a suppression execution even though memory corruption still exists in the execution. Premature termination would mean that the technique would identify some point of memory corruption along the path from the error to the point of the failure, but it would not be the *first* point of memory corruption.

5.3.1 The Variable Re-ordering Algorithm

To address these limitations, the idea of *variable re-ordering* is developed to expose crashes due to memory corruption where crashes may not otherwise occur. Variable re-ordering involves altering the relative ordering of variable locations in memory, prior to using them during execution. The idea is based on the observation that memory errors often lead to unexpected reading or writing of memory locations (other variables) at execution points when those variables should not have been accessed. Depending upon which variables are unexpectedly accessed, a crash may or may not occur. For instance, overflowing a buffer may cause other program variables to be unexpectedly overwritten. As another example, writing to the location pointed to by an infected pointer variable may cause some other variable to be unexpectedly overwritten.

The general algorithm for variable re-ordering to expose crashes is presented in Figure 5.5. The approach is essentially a search algorithm that tries different variable orderings to try to find one that leads to a crash. The input to the algorithm is a non-crashing execution. A second, optional input is a set of memory locations previously known to be corrupt; this information may be available if Execution Suppression was previously run to identify a subset of the corrupted memory, prior to invoking the variable re-ordering algorithm. This optional information can be useful for prioritizing the variables to re-order, to try to expose crashes more quickly. The first step of the algorithm is to identify the set of all variables accessed at some point during the execution (line 1). These variables may be in global space, on the stack, or in the heap. Only the accessed variables need to be considered as candidates for re-ordering, since non-accessed variables in the execution cannot lead to any crashes. Next, the *global*, *stack*, and *heap* memory spaces to consider are

input:
Program execution E that does not result in any crash.
(Optional): set of known corrupt memory locations $Corrupt$.
output:
A variable ordering O that causes execution E to result in a crash, or else NULL if no such O can be found.
algorithm VariableReordering
begin
1: $V_{accessed}$:= set of global, stack, and heap variables accessed during execution E;
2: *Spaces* := {global, stack, heap};
3: *Spaces* := **sort** *Spaces* so those associated with at least one address in $Corrupt$ (if specified) are ordered first; break ties in increasing order of # of associated variables in $V_{accessed}$;
4: **for each** type of memory space *space* $\in$ *Spaces* taken in sorted order **do**
5: $Var_Orderings$:= set of distinct variable orderings involving variables in $V_{accessed}$ that are associated with memory space *space*;
6: **for each** variable ordering $O \in Var_Orderings$ **do**
7: **if** variable ordering O applied to execution E causes E to crash **then report** O;
endfor
endfor
8: **report** NULL;
end VariableReordering

Figure 5.5: General variable re-ordering algorithm to expose crashes in an execution that triggers memory corruption.

ordered so that if any of these spaces involve known corrupted memory, they are ordered first; ties are broken by ordering the spaces in increasing order of the number of accessed variables associated with each space (lines 2–3). Considering the memory spaces in this order (loop in lines 4–7) increases the chances that a crash will be exposed quickly. This is because memory spaces already known to involve memory corruption might be more likely to cause crashes if the variables within these spaces are re-ordered. For the variables in each memory space, the set of distinct variable orderings to try are found (line 5). Heuristics may need to be used here to limit the number of orderings in cases where there may be a large number of potential orderings to try (these heuristics are discussed later in this section).

Next, for each variable ordering, that ordering is applied to the given execution to see if a crash occurs; if so, then the particular ordering causing the crash is reported (lines 6–7). If no crashes occur after trying all variable orderings, then the value NULL is returned (line 8) to indicate that no ordering was found.

Practical Considerations

In practice, there are likely to be many possible variable orderings for a particular execution. For example, for program `xv` used as one of the experimental benchmarks, it turns out that one of the analyzed executions involved accessing over 400 distinct global variables. To blindly try all possible permutations of these global variables in memory would take a very long time. Instead, a heuristic can be used that significantly limits the number of variable orderings to try, while still likely exposing crashes through variable re-ordering when it is possible to do so. This heuristic is based on the observation that crashes are usually caused by infection of address-related variables (either pointer variables, or variables that are used to compute addresses, such as array index variables). Moreover, these variables are most likely to become infected when they are placed immediately after buffers, because potential buffer overflows may unexpectedly overwrite these variables. Thus, the heuristic considers only those accessed variables that are either associated with buffers, or else are used to compute addresses. Further, these variables are re-ordered only to ensure that different address-related variables are placed immediately after different buffers (no need to try all possible permutations of the variables). Thus, variables are re-ordered by laying out different ($buffer$, *address-variable*) pairs in memory. This significantly reduces the number of accessed variables to consider for re-ordering, and drastically reduces the total number

of variable re-orderings that need to be performed. Moreover, each program execution that performs variable re-ordering can account for multiple ($buffer$, *address-variable*) pairs. For example, if there are 5 distinct buffers and 5 distinct address-related variables under consideration, then in total there are 25 ($buffer$, *address-variable*) pairs of interest; however, only 5 executions are needed to account for all of these, since 5 different pairs can be simultaneously handled during each execution.

For each of the global, stack, and heap spaces, different techniques are used to re-order the associated variables. In global space, considered variables are simply rearranged in global memory prior to program execution. One way to implement this would be to modify a compiler to alter the layout of globals in memory. However, the experiments instead used the Valgrind dynamic binary translation framework [55, 103] to simulate this. The implementation allocates custom memory space for global variables and then maps each global variable to a corresponding (specially-ordered) location in the custom memory space. It is ensured that any subsequent accesses to global variables operate on the custom memory space. In stack space, considered local variables at the start of each function call are rearranged on the call stack by instrumenting within Valgrind to modify the order in which they are pushed onto the call stack; references to the local variables within the function call are then adjusted accordingly. For function calls that involve at least one accessed stack buffer, it is ensured that one of the attempted variable orderings involves placing the function call return address immediately after a stack buffer (to expose stack smashing when it is possible). Also, re-ordering of local variables may involve moving them to global space, which is semantically correct as long as a particular function is known to have at most one activation record on the call stack at any given time. Finally, the

problem of variable re-ordering in heap space is more challenging because these variables are dynamically allocated and deallocated during execution. For simplicity, heap variables are specially handled and are not re-ordered in memory. Instead, a special "magic value" is used and positioned adjacent to each heap variable. At the time of variable deallocation or execution termination, the magic value is checked to see if it has been overwritten; if so, a program abort (crash) is produced indicating which program instruction overwrote the magic value.

5.3.2 Example using the Variable Re-ordering Algorithm

Figure 5.6 shows a selection of statements from the `ncompress` program analyzed in the study of memory corruption in Section 5.1. In this program, the stack buffer *tempname* in function `comprexx` is allocated with a fixed size at line 884. Thus, the `strcpy` call at line 886 can overflow this buffer if *fileptr*, which points to an input string of arbitrary length, is too long. For this program, one of the inputs analyzed in the memory corruption study caused the buffer to be overflowed, but resulted in no crash. This is because, on the call stack, the local integer variables *fdin* and *fdout* were positioned in memory directly after the buffer *tempname*, but before the function call return address. In the non-crashing input, it turned out that the overflow was relatively small and only infected the values of the two local integer variables, whose infected values were not subsequently used in a way that could result in a program crash. However, through variable re-ordering, one of the considered alternative orderings is one in which the function call return address is placed immediately after the overflowed buffer on the call stack. Under this variable ordering, the same buffer overflow now corrupts the function call return address, leading to a crash.

```
compress42.c
void comprexx (char **fileptr) {
882:        int fdin;
883:        int fdout;
884:        char tempname[MAXPATHLEN];
886:        strcpy (tempname,*fileptr);        STACK OVERFLOW
}
```

Figure 5.6: Selection of statements from the `ncompress` program to illustrate variable re-ordering.

5.4 The Complete Execution Suppression Technique

Figure 5.7 shows the complete Execution Suppression algorithm incorporating both the suppression technique and the variable re-ordering technique. Note that the required input to this algorithm is simply a program and corresponding input that causes memory corruption. Because of variable re-ordering, it is not necessary for the program input to initially result in a crash. Given a program execution involving memory corruption (line 1), the suppression technique is executed (line 3). Suppression might terminate immediately if the initial execution does not result in a crash. Otherwise, the suppression technique will proceed until an execution results in no crash. In the case that the identified statement is not the true first point of memory corruption, there is a chance that variable re-ordering will expose a new crash. Thus, the variable re-ordering technique is initiated to see whether any further crashes can be found to expose more memory corruption (line 4). If a variable ordering causing a crash is found, then the modified execution with appropriate variable ordering to cause the crash is identified (lines 5–6). This crashing execution is then passed once again to the suppression technique to resume isolating the first point of memory corruption (back at line 3). The algorithm iterates until finally there are no crashes in

input:
  Program P and test case t causing memory corruption due to a memory error.
output:
  Stmt(s) identified as the first point of memory corruption in execution of t on P.
algorithm CompleteExecutionSuppressionTechnique
begin
1: E := the execution of test case t on program P;
2: **do**
3:   $Identified_Statement$:= run suppression technique using E;
4:   $Variable_Ordering$:= run variable re-ordering technique using the most recent suppression execution performed in line 3 above;
5:   **if** ($Variable_Ordering$!= NULL) **then**
6:     E := the new crashing execution using $Variable_Ordering$ computed in line 4 above;
7: **while** ($Variable_Ordering$!= NULL);
8: **report** $Identified_Statement$;
end CompleteExecutionSuppressionTechnique

Figure 5.7: The complete Execution Suppression algorithm to isolate the first point of memory corruption in an execution.

the suppression technique and the variable re-ordering technique cannot find any further crashes. At this point, the most recent statement identified by the suppression technique is reported as the likely first point of memory corruption (line 8). As mentioned earlier, the output of Execution Suppression may include more than one statement in certain cases. However, this situation did not arise in the experiments.

5.5 Evaluation of Execution Suppression

To evaluate the effectiveness and efficiency of Execution Suppression in locating memory errors, experiments were conducted using the same set of benchmark programs and inputs described previously in Tables 5.1 and 5.2 in the memory corruption study in Section 5.1. Implementation details for suppression are described later in Chapter 7.

5.5.1 Experiments with Suppression Only

First, every crashing input was considered and executed using the basic suppression technique *without variable re-ordering*. The results are shown in Table 5.3. For each crashing input, the table shows the total number of program executions required by the suppression technique to isolate the first point of memory corruption ("# Exec. Req."). Also, the statement identified by the technique is reported ("Identified Statement"), along with the maximum static dependence distances from the identified statement to the first point of memory corruption ("1st Corr."), and from the identified statement to the actual memory error ("Error"). For example, in the crashing execution for program `man`, a total of 2 program executions are required by the suppression technique: the first is the original crashing execution, and the second is a suppression re-execution that resulted in no further crashes. A statement was identified that happened to be 1 dependence edge away from the first point of memory corruption, and 2 dependence edges away from the error. Although the first point of memory corruption was missed in this case, the suppression technique was able to precisely identify the first point of memory corruption in the executions for all other inputs (even for programs `ncompress` and `polymorph` where crashes occurred at two different program points). As was observed previously in the memory corruption study and as is shown in Table 5.3, these identified statements were either at, or relatively close to, the memory errors.

Program `man` was the only case in which the suppression technique was not able to precisely identify the first point of memory corruption. In this case, the technique terminated prematurely because no crash occurred even though memory corruption was still present in the execution. Thus, the complete Execution Suppression technique that includes

Program Name	Input Type	# Exec. Req.	Identified Statement	Dep. Distance To...	
				1st Corr.	Error
gzip-1.2.4	Crash Point 1	2	gzip.c: 828	0	0
man-1.5h1	Crash Point 1	2	manfile.c: 243	1	2
bc-1.06	Crash Point 1	2	storage.c: 177	0	1
pine-4.44	Crash Point 1	2	bldaddr.c: 7126	0	5
mutt-1.4.2.1	Crash Point 1	3	utf7.c: 192	0	1
ncompress-4.2.4	Crash Point 1	2	compress42.c:886	0	0
	Crash Point 2	4	compress42.c:886	0	0
polymorph-0.4.0	Crash Point 1	2	polymorph.c:198	0	1
	Crash Point 2	3	polymorph.c:193	0	1
xv-3.10a	Crash Point 1	4	xvbmp.c: 167	0	2
tar-1.13.25	Crash Point 1	2	incremen.c: 180	0	0
tidy-34132	Crash Point 1	2	parser.c: 854	0	0
cvs-1.11.4	Crash Point 1	2	server.c: 992	0	0

Table 5.3: Experimental results using only suppression (no variable re-ordering), with respect to different crashing inputs on the benchmark programs.

variable re-ordering was used to see if the results for `man` could be improved. Indeed, as is shown in Table 5.4, variable re-ordering allows the first point of memory corruption to be precisely identified in the execution for `man`. This is because variable re-ordering exposes one additional crash in the execution that was not originally observed when running the suppression-only technique. In this case, forcing an array index variable to be located in memory directly after an overflowed buffer causes a new crash to occur, revealing some additional corrupted memory that allows the technique to precisely find the first point of memory corruption.

5.5.2 Experiments with Suppression and Variable Re-ordering

An important feature of variable re-ordering is that it enables Execution Suppression to be applicable to other inputs that trigger memory corruption but do not initially result in a program crash. Thus, besides showing the revised results for program `man`, Ta-

Program Name	**Input Type**	**# Crash Exposed**	**# V.O. Exec.**	**Identified Statement**	**Dep. Dist. To...**	
					1st Corr.	**Error**
gzip-1.2.4	No Crash	0	15	—	—	—
man-1.5h1	Crash 1	1	18	man.c: 977	0	1
bc-1.06	No Crash	1	—	storage.c: 177	0	1
pine-4.44	No Crash	1	—	bldaddr.c: 7126	0	5
mutt-1.4.2.1	No Crash	1	—	utf7.c: 192	0	1
ncompress -4.2.4	No Crash	1	5	compress42.c:886	0	0
polymorph -0.4.0	No Crash	1	6	polymorph.c:198	0	1
xv-3.10a	No Crash	1	135	xvbmp.c: 167	0	2

Table 5.4: Experimental results using suppression and variable re-ordering (the complete Execution Suppression technique), using different inputs on the benchmark programs.

ble 5.4 shows the results of running the complete Execution Suppression technique using the seven non-crashing inputs analyzed earlier in the memory corruption study in Section 5.1. Without variable re-ordering, it would not have been possible to have applied Execution Suppression to these non-crashing inputs.

The format of Table 5.4 is the same as Table 5.3, except the column "# Exec. Req." has been replaced by two columns: "# Crash Exposed", indicating how many additional crashes were found through the use of variable re-ordering before the technique terminated (1 in all cases except for `gzip`); and "# V.O. Exec.", indicating the maximum number of variable re-ordering *executions* required in order to find the exposed crash (in the case of `gzip`, the number of re-ordering executions in order to discover that no crash could be exposed). In the table, the number of required variable re-ordering executions is not listed for programs `bc`, `pine`, and `mutt`. This is because these three benchmarks involve heap-buffer overflows, and as was described previously, heap buffers are handled differently and variable re-ordering is not performed with heap variables.

It turns out that for all of the non-crashing inputs except `gzip`, it was possible to

find a particular variable re-ordering that exposed a crash in the execution. In all of these cases, this made suppression applicable, which in turn resulted in the first point of memory corruption being precisely identified without the need to expose any more crashes through variable re-ordering. For `gzip`, the initial input did not crash because a global buffer was overflowed by 1 position, erroneously writing the value NULL into another global variable that happened to already have value NULL. In this case, no memory corruption actually occurred during execution because a memory location was overwritten with the *same* value. For this execution, none of the other global variables were accessed in the rest of the execution in such a way that they could have resulted in a crash if they had been infected. As a result, the variable re-ordering technique could not force a crash to happen for this particular case.

In order to expose a crash through variable re-ordering, the maximum number of program executions required to achieve this for each benchmark ranged from 5 executions for program `ncompress`, to 135 executions for program `xv`. The reason `xv` requires so many distinct program executions in this case, is because the particular execution under consideration happens to access 202 different global buffers and 67 different address-related variables (much higher than in all the other benchmarks). It therefore takes quite a few executions in this case to group these variables in different ways to try to expose crashes.

Across all of the benchmark programs, the actual time required to execute each program given the suppression and variable re-ordering implementations never took more than a few seconds per execution. When performing only suppression, the experiments required between 2 and 4 executions per benchmark, so the total time required to run the technique was only a matter of seconds for each program. However, when using variable re-

ordering, many more program executions may be performed when trying different variable orderings to expose crashes. In the experiments, this translated to total runtime ranging from several seconds (for `ncompress` and `polymorph`), to several minutes (`xv` was the worst case that required about 5 minutes to run). This timing may be reasonable in a debugging context.

5.6 Summary

In this chapter, the state alteration technique called Execution Suppression was developed for assisting in the task of locating memory errors in software. The technique uses the notion of suppression to iteratively identify and avoid the effects of known corrupted memory locations in a crashing execution, until the first point of memory corruption in the execution can be identified. This point is likely to be at, or near to, the error. It was shown how the idea of variable re-ordering can be used to expose crashes due to memory corruption in cases where crashes may not otherwise occur. By combining the ideas of suppression and variable re-ordering, Execution Suppression can become highly effective at assisting in the location of memory errors.

To motivate the development of Execution Suppression, a detailed study was presented of 11 real-world benchmark programs containing known memory errors that involve varying degrees of memory corruption propagation. An experimental analysis of Execution Suppression was conducted using the 11 benchmark programs. In all cases, the technique was able to precisely identify the first point of memory corruption in an execution, and this point was always either at, or very close to, the memory error.

Several observations can be made about Execution Suppression. First, the idea

of suppression is effective because it works in the general case when memory corruption propagation occurs in a distributed fashion – with each infected memory location potentially influencing multiple other memory locations – rather than in a straight-line fashion. Even though suppressing some corruption may avoid one failure, there is a chance that any remaining corruption would still lead to subsequent failures. The technique can also be effective when multiple independent memory errors exist in a program simultaneously. On each iteration of the technique, the algorithm gets closer to identifying the first point of memory corruption for *some* memory error (the technique is not sensitive to *which* memory error). Once a first point of memory corruption is found, the associated error can be fixed and then the technique can be run again on the modified program to identify any remaining memory errors.

Execution Suppression can also be used to locate a variety of memory errors. This is because the technique views memory errors in terms of accesses to corrupted memory locations during program execution, and this trait is shared by most memory errors. For example, consider a dangling pointer to a deallocated memory location. Suppose this memory location is later re-allocated and used, but in the meantime (due to the dangling pointer), an unexpected write occurs to this memory location, causing a crash once the infected value at that location is accessed. In general, this type of error can be particularly tricky to find, especially since the offending write can be associated with a completely different type than the type associated with the access that causes the crash. However, in this situation the technique can immediately identify the last instruction performing the offending write due to the dangling pointer.

One limitation of Execution Suppression is that it assumes program crashes are

caused by memory corruption. However, this is not always the case, and the technique is not designed to handle memory errors and crashes that do not involve memory corruption. An important class of such memory errors is the *memory leak*, in which allocated memory is not deallocated when it is no longer needed. This can eventually cause a crash when the program runs out of available memory. However, memory leak errors do not involve memory corruption, according to the definition. Thus, Execution Suppression is expected to be ineffective for locating the root causes of memory leaks.

A lingering question about Execution Suppression is how it will work for multithreaded programs. This is an important question because memory errors can exist in both single-threaded and multithreaded programs. It turns out that the basic Execution Suppression technique does not address the unique traits of a very important type of multithreading error that can involve memory corruption: the data race error. In the next chapter, it is shown how the notion of suppression can be extended to handle multithreading errors including data races.

Chapter 6

Execution Suppression and Multithreading Errors

In the previous chapter, the Execution Suppression technique was presented that can be used to isolate memory corruption to locate memory errors. The technique iteratively examines program crashes due to memory corruption to identify subsets of memory corruption involved in a program execution, until the first point of memory corruption can be identified. However, the technique implicitly assumes that the faulty program being analyzed is single-threaded and deterministic, i.e., the same failure will occur on multiple program executions using the same input values. The effectiveness of the technique on multithreaded programs is limited for two important reasons. First, the technique does not include a mechanism to ensure that a failure due to a multithreading error can be repeated on subsequent executions. In general, multithreaded programs can execute non-deterministically because different thread interleavings can occur on different program executions, even though the executions may involve the same set of input values. In the presence of multithreading

errors, such non-determinism may cause a failure to manifest on only some executions but not others. Since Execution Suppression is an iterative technique that performs multiple program executions using the same set of input values, a mechanism is required to ensure that the same thread interleaving occurs on multiple program re-executions. The second reason why the effectiveness of the technique is limited for multithreaded programs is because the technique does not account for the unique traits of *data races*, an important class of multithreading errors that is caused by an unexpected interleaving of parallel threads and can result in memory corruption. In this chapter, it is shown how Execution Suppression can be extended [70] to locate multithreading errors that involve memory corruption.

Execution Suppression considers read-after-write (RAW) dependencies when determining which statements are affected by memory corruption during execution and should be suppressed. As was observed by Tallam et. al. [130], two additional kinds of dependencies should also be considered when detecting data races: write-after-read (WAR) and write-after-write (WAW) dependencies. The extended technique accounts for these other dependencies in order to be able to identify data race errors on-the-fly. Prior work on classifying benign and harmful data races [101] has also shown that not all data races are harmful, i.e., they may not always affect the correctness of a program execution. For instance, sometimes data races are involved when synchronizing threads in multithreaded executions, but these data races do not negatively affect the correctness of a program. These kinds of data races should not be identified as the likely root cause of a failure. Thus, this has inspired the development of a new component in Execution Suppression that checks whether a data race is potentially harmful before reporting it to a developer as the likely root cause of a failure.

6.1 Locating Multithreading Errors using Suppression

The revised technique for locating multithreading errors can work for data race errors, in addition to the other memory errors involving memory corruption that can be handled by the basic technique presented in Chapter 5. This includes errors such as buffer overflows (including corruption of function call return addresses), uninitialized reads, dangling pointers, and double frees. The same definition of memory corruption is used as was presented in Section 5.1.2.

A *data race* in an executing program occurs when multiple parallel threads perform unsynchronized accesses to shared data. These data races can potentially be harmful, i.e., force the execution into an unexpected state, when at least one of those accesses is a write to a shared memory location. Intuitively, a data race can lead to such problems as a value being unexpectedly overwritten or a stale value being read. The technique is designed to dynamically check for potentially-harmful data races that may cause a failure during execution.

Definition 10 (Data Race).

Given a multithreaded program execution, a **data race** *can be defined as the concurrent access of a shared memory location by two or more different threads, such that the following two conditions hold: (1) at least one of those accesses involves a write to that memory location; and (2) there is no synchronization specified between those memory accesses.*

As discussed in Chapter 5, memory corruption resulting from an error can propagate (spread) to other memory locations until finally a subset of the corruption may directly cause a program failure. In a multithreaded program, the error that is the root cause of

such a memory-related failure can either be a data race, or else some other kind of memory error. The extended version of Execution Suppression iteratively isolates the error through multiple program executions. The technique is designed to specially alter the state of each execution through suppression to reveal more of the memory corruption, until finally the root cause is revealed. If any execution results in no failure, then the technique terminates and assumes that the root cause has been found. The technique also terminates if a potentially-harmful data race is found to be directly involved in the failure, as the data race is assumed to be the root cause. Note that a program failure can occur in the form of a program crash, or else any other observable memory-related failure such as incorrect output that could indicate memory corruption.

Each iteration of the technique analyzes a program failure as follows. First, the technique determines whether a potentially-harmful data race is directly involved in the failure. If so, the data race is reported and the technique terminates. If not, then the memory corruption that directly caused the failure is *suppressed* during the execution, following the same semantics as was previously described for the basic Execution Suppression technique in Chapter 5.

The overall technique is composed of three main tasks: (1) ensuring that a multithreaded error can be faithfully traversed on multiple program executions; (2) determining on-the-fly whether a particular statement instance during execution is involved in a potentially-harmful data race; and (3) performing suppression in order to account for propagation of memory corruption during execution. The technique is presented in Figure 6.1. The approach takes as input a single-threaded or multithreaded program and an associated test case that causes a memory-related failure in the program. The output of the approach

input:

Single-threaded or multithreaded program P and test case t causing a program failure.

output:

Data race or statement(s) identified as the root cause of the failure.

algorithm MultithreadedExecutionSuppressionTechnique

begin

// Task 1: Ensure effect of error can be reproduced on multiple executions.

1: **execute** P using t and analyze the execution to ensure that the failure can be faithfully reproduced;

2: S_{def} := "undefined";

3: S_{supp} := {};

4: **while** an observable memory failure f occurs during execution of P using t **do**

5: S_{use} := the statement instance at which f occurs;

6: L := the used memory location(s) causing f at S_{use};

7: S_{def} := the statement instance(s) defining the value(s) in L prior to its use at S_{use};

8: **if** S_{def} does not exist **then**

9 S_{def} := the statement instance(s) defining the address(es) of L prior to its use at S_{use};

endif

// Task 2: On-the-fly check for WAR and WAW potentially-harmful data race.

10: **re-execute** P using t until S_{def} is reached, while monitoring accesses to L (for WAR and WAW data race checking);

11: **if** $\exists$ WAR or WAW data race D at S_{def} **then**

12: **if** D is potentially harmful **then output** D and **return**;

endif

// Tasks 2 and 3: Perform execution suppression while also checking on-the-fly
// for RAW potentially-harmful data race.

13: **resume** execution from S_{def} while (1) suppressing:

(a) the definition of L at S_{def};

(b) statement instances in S_{supp}; and

(c) any subsequent statement instances directly or indirectly influenced by the definition of L at S_{def};

and (2) monitoring accesses to L (for RAW data race checking):

if $\exists$ potentially-harmful RAW data race D **then output** D and **return**;

14: **augment** set S_{supp} with the additional statements suppressed at line 13 above;

endwhile

15: **report** the statement(s) associated with the latest S_{def};

end MultithreadedExecutionSuppressionTechnique

Figure 6.1: The extended Execution Suppression algorithm for locating multithreading errors.

is either a potentially-harmful data race, or else a program statement that is identified as the likely root cause of the failure. As was the case for the basic Execution Suppression technique, more than one program statement may be identified by the technique on some occasions. Each of the three main tasks of the technique are now described, with respect to how they appear in Figure 6.1.

6.1.1 Reproducing the Effects of Multithreading Errors

When debugging any program, it is necessary to be able to reproduce a program failure so that a failing execution can be analyzed to deduce what is going wrong and to figure out how to fix the problem. For single-threaded, deterministic programs, it is enough to simply re-execute the program using the same set of input values. However, for multithreaded programs, being able to repeat a failure is generally not a trivial task. This is because certain multithreading errors may only manifest themselves under particular thread interleavings, and these interleavings may change between executions even when those executions are based on the same input values. Since the technique involves multiple program executions, it is required that each execution be a faithful reproduction of the original failing execution (not counting the alterations to the execution state performed by the technique). The technique must address this issue in order to be applicable to multithreaded programs.

One way to accomplish this repeatability would be to use a logging/checkpointing system. This can be used to effectively record an execution so that it can be faithfully replayed. One such system is *jockey* [123], a user-level library for performing checkpointing and logging for the purpose of replaying a program execution. Another approach to ensure

repeatability would be to control the scheduling of threads during execution, to ensure that the same sequence of thread interleavings is executed each time the program is run on the same input [98]. Valgrind [55, 103] is a user-space dynamic binary translation infrastructure that can be used to perform this task. Valgrind provides a synthetic CPU in software for program execution, and includes its own thread scheduling mechanism. This scheduler can be modified to force a particular thread interleaving during execution. The experiments used this Valgrind-based implementation. In general, the implementation would need to account for other non-deterministic factors that may be involved in the execution besides thread interleaving (such as reading input values from the execution environment). However, this was not required for the experimental benchmarks.

In the algorithm in Figure 6.1, the task of ensuring faithful reproduction of the effects of multithreading errors is performed at line 1, before the main iterative loop begins. The subject program is executed to produce the failure, and during execution, the information necessary to ensure faithful reproduction of the execution is recorded. This information is then used on each iteration of the technique to ensure that the effect of any multithreading errors is faithfully repeated.

The main loop of the technique is shown in lines 4 – 14 in Figure 6.1. As long as the program execution ends in a failure (loop condition at line 4), the technique analyzes the executed statement instance that is *directly involved* in the failure. Identifying this statement instance is straightforward because the point of the failure is known, and therefore the accessed memory location(s) causing the failure can also be easily found (lines 5–7). As shown in lines 7–9, the technique targets the last definition (write) that directly led to the failure. This statement instance may or may not be actual root cause of the failure. One

way the statement instance is likely to be the root cause, is if the instance is associated with a potentially-harmful data race occurring in the execution.

6.1.2 On-the-fly Checking for Data Races

The approach for on-the-fly checking for data races is composed of two main steps. First, the technique checks for the existence of a data race. Second, the technique determines whether the identified data race is potentially harmful. Only identified data races that are potentially harmful are reported by the technique.

Identifying a Data Race

Given a target statement instance to analyze such that it performs a write, the technique first determines whether a data race exists at this statement instance as follows. At this statement instance, it is known which thread executed the write. Thus, when re-executing the program, the technique monitors on-the-fly any access to this memory location that comes from a different thread. In particular, the technique finds the last access (read or write) to this memory location that comes from a different thread, and checks for any synchronization that may exist between these two accesses. If no synchronization exists, then this implies that there is either a write-after-read (WAR) or a write-after-write (WAW) dependence between these two memory accesses that represents a data race. This step is performed in line 10 in Figure 6.1. If there is no WAR or WAW data race here, then the execution continues from this point while the technique monitors for any read access to the same location coming from a different thread. Again, if there is no synchronization between two such accesses, this indicates a read-after-write (RAW) data race. This step is performed

along with suppression at line 13 in Figure 6.1.

Figure 6.2 shows the pseudocode for this on-the-fly method of detecting data races. This accounts for the observation by Tallam et. al. [132] that data races can be represented by either RAW, WAR, or WAW dependencies. Note that in Figure 6.1, the checking for RAW data races is done on-the-fly while also performing the suppression. At any time during the rest of the execution, if a RAW data race is found and then determined to be potentially harmful, the technique immediately reports it and then terminates.

The technique for detecting a data race guarantees that the data race exists, because it is detected as having actually occurred during a program execution. However, a data race may or may not be *potentially harmful* [101]. For example, data races that do not affect the state of the executing program are not harmful. The technique reports a data race only if it is determined to be potentially harmful, because only then can it possibly be the root cause for a failure.

Determining Whether a Data Race is Potentially Harmful

The technique determines whether a data race is potentially harmful by altering the sequence of executing threads. Suppose that the two memory accesses involved in a data race are at points $access_1$ and $access_2$ during execution, and these accesses are performed by threads t_1 and t_2, respectively. Suppose further that at point $access_1$, the set T includes all threads that are ready to execute *except* t_1. Also assume that at point $access_2$, the value stored in the associated memory location is v. Then the technique executes the program up to $|T|$ times; on each execution, a different thread from T is allowed to execute at point $access_1$ in place of thread t_1. Then at the point during execution when the second memory

```
input:
    Execution E with target write instruction instance w, writing to location l and
      executed by thread t.
output:
    An identified data race, or NULL if no data race is found.
algorithm SearchForDataRace
begin
1:  foundSync := false;
2:  lastAccess := "undefined";
    // Monitor for WAR and WAW data races.
3:  for each statement instance i in execution E do
4:    if i == w then
5:      if lastAccess != "undefined" && foundSync == false then
6:        output data race between lastAccess and w and return;
        else
7:        break;
        endif
8:    else if i accesses location l using thread other than t then
9:      lastAccess := i;
10:     foundSync := false;
11:   else if i performs thread synchronization related to accessing location l then
12:     foundSync := true;
      endif
    endfor
    // Monitor for RAW data races (if no other races found above).
13: for each remaining statement instance i in execution E do
14:   if i reads from location l using thread other than t then
        // If we reach here, there is no synchronization, otherwise the program
        // would have returned below at line 17.
15:     output data race between i and w and return;
16:   else if i performs thread synchronization related to accessing location l then
        // There cannot be a RAW data race.
17:     report NULL and return;
      endif
    endfor
18: report NULL;
end SearchForDataRace
```

Figure 6.2: On-the-fly checking for data races.

access occurs, the technique simply checks whether the value contained in the associated memory location is different than v (has been changed). If so, the data race is determined to be potentially harmful. If the value is not changed in any of the $|T|$ executions, then the data race is determined to *not* be potentially harmful. As shown in lines 12 and 13 in Figure 6.1, a data race is reported only if it is determined to be potentially harmful.

6.1.3 Performing Suppression

The final task performed by the technique is to carry out suppression, which follows the same semantics as was described for the basic Execution Suppression technique in Chapter 5. Suppression occurs in the event that the store statement instance directly causing a failure is not found to be associated with any WAR or WAW potentially-harmful data race. It is performed simultaneously with monitoring for a RAW data race during execution (line 13 in Figure 6.1). If no potentially-harmful RAW data race is found during suppression, and no additional failures occur during the execution as a result of the suppression, then the most recently-suppressed statement instance is likely to be associated with the error. It is therefore reported at line 15 in Figure 6.1.

6.2 Illustrative Examples

6.2.1 Example with Harmful Data Race Error

Consider the example multithreaded code snippet shown in Figure 6.3. In this piece of code, assume that x and y are pointers to shared memory locations, each containing an integer. Further assume lines 1 – 2 are erroneously not protected in a critical section, despite the fact that they involve a read and subsequent write to shared location x. A

```
Let x and y be pointers to shared memory locations.
Let foo, foo2 be functions that take an integer input and return an integer.
Let structArray, structArray2 be heap arrays of pointers to structs with a
    field named data.

// Assume *x has been initialized to some value.
// x not protected by critical section in lines 1–2.
1:  int a = foo(*x);
2:  *x = *x + a;
    start_critical_section();
      // Defined value of y may be incorrect.
3:    *y = foo2(*x);
4:    int b = structArray[*y]->data;
    end_critical_section();
5:  int c = structArray2[*x]->data;
```

Figure 6.3: Example multithreaded code snippet, part 1.

potentially-harmful data race exists at this point, and this is the error that needs to be located by the technique. Lines 3 – 4, which involve a write and subsequent read to shared location y, are protected by a critical section. However, the write into location y at line 3 may write the wrong value, due to the data race in lines 1 and 2 involving location x. Since $*y$ is then used as an index into an array in line 4, there is potential for a crash at this point. Similarly, there is potential for a crash at line 5, which uses $*x$ as an array index.

Suppose that this example code is executed using two parallel threads, and that this execution ends in a failure (crash), as depicted in Figure 6.4 (A). In this execution, suppose thread 1 executes line 1 first, then this is immediately followed by thread 2 executing line 1 (this is possible since lines 1 – 2 are not protected by synchronization). In this case, thread 2 reads a stale (incorrect) value from location x, since it was expected that this read would not occur until after thread 1 updates x in line 2. Next, suppose thread 2 writes to x in line 2; this writes an unexpected value to x since this value is computed using the

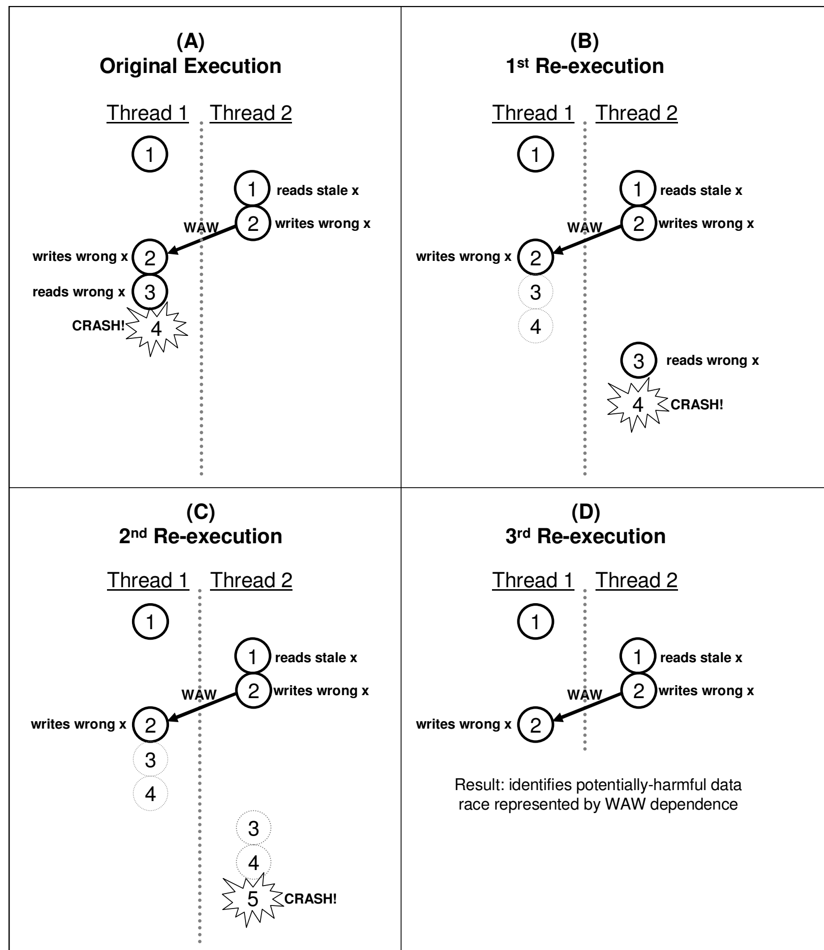

Figure 6.4: Example executions for running technique with a harmful data race error, to accompany Figure 6.3.

stale value for x read from the same thread in line 1. Now, suppose thread 1 resumes and writes to x in line 2; again, this writes an incorrect value to x since it is computed using an unexpected value for x obtained from the write in thread 2. If thread 1 continues executing, line 3 reads the wrong value for x and then defines an incorrect value for y, which is then used in line 4. Assume that at line 4, execution crashes due to a buffer overflow.

Since the array index $*y$ is directly related to the crash at line 4, the technique determines that the last definition of this location occurred at line 3. The next execution of

the code is shown in Figure 6.4 (B). During this execution, it turns out that there is no data race involved at line 3. This is because there are no prior accesses to location y before line 3 (so it is not involved in any WAR or WAW dependence), and there is no subsequent read to location y after line 3 from another thread that is not protected by synchronization. Thus, the definition of y at line 3 is suppressed, and the definition of b at line 4 is also suppressed since it uses the suppressed definition at line 3. Thus, the original crash is avoided. Now, suppose execution continues as thread 2 resumes and executes lines 3 and 4 (these lines are not suppressed in this execution since they do not depend on the previously-suppressed instances of lines 3 and 4 from the other thread). Suppose again that there is a crash at this new instance of line 4 in thread 2. Again, the last definition of y is at line 3. During the next execution, shown in Figure 6.4 (C), there is again no data race detected at this point. Lines 3 and 4 are then suppressed in the execution of thread 2. Now, execution is able to reach line 5 in thread 2. Suppose that a crash occurs here due to the use of incorrect $*x$ as an array index.

The technique determines that the last definition to location x occurred at line 2 of thread 1. The next execution is shown in Figure 6.4 (D). When control reaches line 2 of thread 1, a data race check is performed. In this case, it is discovered that the last access to location x by another thread is the write in line 2 of thread 2. Since there is no synchronization specified between these two memory accesses, the technique identifies a data race at this point (WAW dependence). Next, the technique checks whether this data race is potentially-harmful. In the original execution, thread 2, line 2 is executed *before* thread 1, line 2. However, during the interval when thread 2 is first executed, thread 1 is also ready to resume execution. Thus, the technique re-executes the program and forces thread 1 to

resume execution at the point in which thread 2 originally would have been scheduled. This effectively alters the interleaving of these two threads so that the program behavior matches what is expected. As a result, the value written into location x at the second access is now changed, and the data race is determined to be potentially harmful. Intuitively, because this execution can produce different values under different thread interleavings, this is likely to be unexpected behavior and is considered to be potentially harmful. The data race error is therefore reported to the user.

6.2.2 Example with Non-Data-Race Error

Suppose the example code from Figure 6.3 is slightly modified to obtain the example code in Figure 6.5. This example is identical to the previous example, except that in this case, lines -1 and 0 have been added to the beginning of the code snippet, and line 6 has been added to the end of the code snippet. Line -1 initializes a variable z to an incorrect constant value (this is the error), which in turn is used in line 0 to initialize x to an incorrect (also constant) value. Line 6 involves an assertion that ensures z contains the correct value; if not, then the program terminates with a failure at this point. Although the error in line -1 is not a memory error, it can lead to an observable failure at line 6, and so can possibly be located by the technique.

Note that the same data race exists here as in the previous example. However, assume in this case that for the constant value used to define location x in line 0, the value returned by the call to foo in line 1 is actually 0. Then in this situation, the data race left over from the previous example is *not* potentially harmful. This is because the value written into location x at line 2 does not change the constant value $*x$, and this is true for

```
Let x and y be pointers to shared memory locations.
Let foo, foo2 be functions that take an integer input and return an integer.
Let structArray, structArray2 be heap arrays of pointers to structs with a
    field named data.

// Assume z is initialized to an incorrect constant.
-1: int z = some_incorrect_constant_value;
0:  *x = 2 * z;
// x not protected by critical section in lines 1–2.
// Assume a is defined to be 0 given constant *x.
1:  int a = foo(*x);
2:  *x = *x + a;
    start_critical_section();
      // Defined value of y may be incorrect.
3:    *y = foo2(*x);
4:    int b = structArray[*y]−>data;
    end_critical_section();
5:  int c = structArray2[*x]−>data;
6:  assert_correct_value(z);
```

Figure 6.5: Example multithreaded code snippet, part 2.

all threads, regardless of the thread interleaving. Thus, in this case, the data race cannot affect the correctness of the program.

Now consider running the technique on this modified example. Suppose that the original execution and the first two re-executions are the same as how they appear in Figure 6.4 (A–C) (with the only minor difference being that lines -1 and 0 now precede execution of line 1 in each of the two threads). The crashes at lines 4 and 5 from the previous example can still occur in the current example because location x still contains an incorrect value in the current example.

Suppose the technique arrives at the end of the second re-execution as shown in Figure 6.4 (C). Then the crash at thread 2, line 5 directly depends upon the store to location x in thread 1, line 2. The next execution, shown in Figure 6.6 (A), is where the behavior

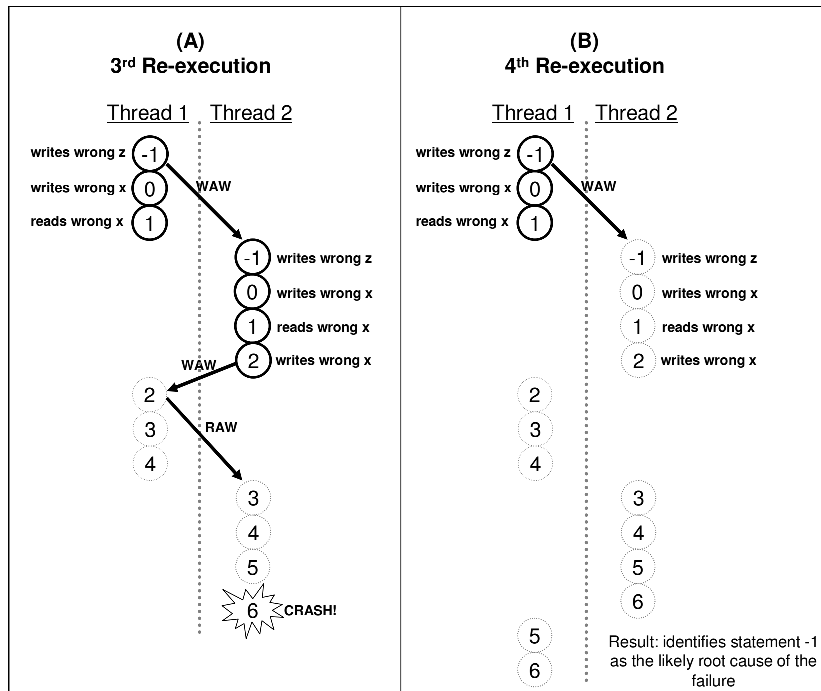

Figure 6.6: Example executions for running technique with a non-data-race error, to accompany Figure 6.5. Assume that the original execution and the first 2 re-executions are similar to those in Figure 6.4 (A–C).

of the technique begins to differ from the previous example. In this execution, control reaches thread 1, line 2. The same data race as was found in the previous example is also found here. However, in this case the data race is found to *not* be potentially harmful, as was explained earlier. Similarly, another data race is found here, which is a RAW data race involving the write in thread 1, line 2, and the subsequent read in thread 2, line 3. However, again this data race is found to *not* be potentially harmful, since the write to location x in line 2 does not actually change its value. Thus, no potentially-harmful data races are identified here, so the technique continues by suppressing the definition of x at thread 1, line 2, and anything dependent upon it. This allows execution of thread 2 to arrive at line 6, which crashes because the sanity check for the value of variable z fails (since z contains

an incorrect value). The last definition of variable z occurs at thread 2, line -1.

In the next execution, shown in Figure 6.6 (B), control arrives at thread 2, line -1. In this case, there exists a previous access to location z from the other thread (thread 1, line -1), so there is a data race involved here represented by a WAW dependence. However, since z is initialized to a constant value, again this data race is not potentially harmful. There are no additional subsequent reads of z prior to the point of the crash, so there are no RAW data races at this point. Thus, the technique continues by suppressing the definition of z in thread 2, line -1, and all of the subsequent statements that are dependent upon it (including all of the statements suppressed in previous iterations). In this example, all remaining executed statements in both threads are suppressed, resulting in no additional program failures. The technique then terminates and identifies statement -1, the statement associated with the most recent point of suppression, as the root cause of the failure.

6.3 Evaluation using Multithreading Errors

6.3.1 Setup for Experiments

To study the effectiveness of the technique for locating errors in multithreaded programs, a set of multithreaded benchmark programs and their associated errors were selected as shown in Table 6.1. This set of programs involves three different data race errors (one in `apache` and two in `mysqld`), as well as four other memory errors (one uninitialized read in `mysqld`, as well as three stack buffer overflows in programs `prozilla` and `axel`). These benchmark programs were selected because they are multithreaded, real-world programs that are generally well-known and widely-used, and they contain real errors. For each of

Program Name	# Lines of Code	Program Description	Root Cause of Failure
apache	191 K	HTTP server (ver. 2.0.48)	data race [90]
mysqld-1	508 K	database server (ver. 3.23.56)	data race [52]
mysqld-2	508 K	database server (ver. 3.23.56)	data race [53]
mysqld-3	508 K	database server (ver. 3.23.56)	uninitialized read [51]
prozilla-1	16 K	download accelerator (ver. 1.3.5.1)	stack buffer overflow [62]
prozilla-2	16 K	download accelerator (ver. 1.3.5.1)	stack buffer overflow [50]
axel	3 K	download accelerator (ver. 1.0a)	stack buffer overflow [63]

Table 6.1: Benchmark programs used in the experiments and their associated errors.

the 7 errors listed in the table, an execution was identified that would traverse the error and trigger an execution failure. This execution was provided as input to the multithreaded Execution Suppression technique to try to isolate the error.

The implementation is primarily written in `C` within the Valgrind infrastructure [55, 103], which allows a program to be dynamically instrumented and modified at runtime. It is assumed that the multithreaded programs under consideration are run on a single-processor system, so that only one thread is running at any given time. Also, it is assumed that the failures triggered in the benchmark programs occur as either program crashes or assertion violations, because this allows the technique to easily and automatically determine whether or not an observable failure occurs on each program execution. Finally, it is assumed that synchronization points are clearly marked in the program so that they can be easily identified. Further implementation details are described in the next chapter (Chapter 7).

6.3.2 Results and Discussion

The results for each of the 7 analyzed errors are shown in Table 6.2. In this table, the middle column shows the total number of executions required by the technique to isolate the error. This number is broken down into three components: the original execution to trigger the failure ("orig," which is always 1); the number of suppression executions required to detect data races and/or carry out suppression ("supp"); and the number of re-executions required to force a different thread interleaving to check whether a detected data race is potentially harmful ("race_check"). The right-most column mentions whether the error is identified by the technique. The results for each of the benchmark programs are now described in detail.

`Apache`. In this benchmark program, there is a data race error in which there is a write to a buffer, followed by an update to a buffer count variable. These two memory accesses involve a shared variable that is not protected in a critical section. Further, this piece of code is related to writing information to a server log. In the event that multiple requests are writing to the server log simultaneously, it is possible that the data in the server log will get corrupted due to the data race. In this case, the error can be represented

Program Name	**Total # Executions Required (orig + supp + race_check)**	**Identified error?**
apache	3 (1 + 1 + 1)	yes
mysqld-1	3 (1 + 1 + 1)	yes
mysqld-2	3 (1 + 1 + 1)	yes
mysqld-3	2 (1 + 1 + 0)	yes
prozilla-1	2 (1 + 1 + 0)	yes
prozilla-2	4 (1 + 3 + 0)	yes
axel	3 (1 + 2 + 0)	yes

Table 6.2: Experimental results of running the technique to locate multithreading errors.

by either a WAR or a WAW data race associated with the unprotected instructions.

When the technique is run on this program using two conflicting requests that trigger the bug, an assertion failure occurs when the server log is found to be corrupted. The associated instruction instance of the corrupting write to the log is identified. This instruction instance uses two variables, a pointer to the buffer itself, as well as a count variable. Since either of these can be corrupted, the technique identifies the last instruction instances in the execution in which these two variables were defined. It turns out the count variable was last defined in the update instruction mentioned above that is associated with a data race. The technique then performs the first re-execution in which data race detection and/or suppression will be carried out. During this execution, memory accesses are monitored for possible data race detection, and the first target instruction reached during the execution is the update to the buffer count variable. At this point, the approach determines that the last access to this shared variable by a *different* thread occurs at the same instruction (but an earlier instance), representing a WAW dependence. Further, there is no synchronization between the executions of these two accesses, so a data race occurs here. The technique then determines that at the point of the first memory access, there is one other thread that is ready to be executed (besides the thread that actually did execute). Thus, the technique performs one more execution and forces the different thread interleaving to occur, and determines that the final value of the shared variable in question, has changed at the point of the second memory access. As a result, the technique determines that the identified data race is potentially harmful, and reports it to the user. This data race is in fact directly associated with the expected root cause of the failure. In total for this benchmark program, there were 3 required program executions: the original execution to

repeat the failure, the first suppression execution in which a data race was detected, and a final execution to determine that the data race was in fact potentially harmful.

`Mysql-1`. In this program, there is an error in which an update to the database state and the associated update to the log file are not protected in a critical section. A race condition may therefore occur when two requests are being handled simultaneously, which can have the effect that the order of entries in the log file may not match the actual order in which updates occurred to the database.

When the technique is run on this program using two simultaneous requests that trigger the bug, an assertion failure occurs when it is determined that the log entries have become out of order. For this benchmark program, the log itself is treated as shared memory, so the technique identifies the last write to the log that caused the out-of-order entry. It is assumed that this write has been corrupted in some way. On the next execution, execution proceeds while monitoring accesses to the log for possible data race detection. Once the target instruction instance is reached at which the log is written, the technique determines that the last access to the log from a different thread occurs when reading from the log's base address, performed by the conflicting thread which is also accessing the log. Further, there is no synchronization specified between these two log accesses, so a WAR data race is detected. It is then determined that at the point of the first memory access, there is one other thread that could have been scheduled instead. As a result, the program is re-executed while forcing the other thread interleaving to occur, and the log is determined to have changed as a consequence. Thus, the associated potentially-harmful data race, which is the root cause of the failure, is reported to the user. As was the case for program `apache`, this subject program requires 3 executions in total.

`Mysql-2`. In this program, there is an error in which a close and subsequent open of a log file is not protected in a critical section. Elsewhere in the code, there is a check on the log file to verify that it is open, before writing to it; if the log is not open, no writing to the log is performed. It is possible with two conflicting threads that one thread closes the log file, then another thread performs updates and does *not* write to the log file (because the log is closed), and finally the original thread re-opens the log file. The effect is that some database updates may be silent and will not be recorded in the log.

When executing the technique on this program, an assertion failure occurs when it is determined that the log file is closed when it was expected to be open. This is represented by a read from the log (which we treat the same as shared memory). The technique determines that the last write to the log in the execution occurred when the log file was closed. On the next execution of the technique, execution proceeds until it reaches the point at which the log file is closed. At this point, it turns out that there are no prior accesses to the log from a *different* thread (only from the same thread), so there are no WAR or WAW dependencies here. Thus, execution continues from this point while performing suppression and also monitoring accesses to the log for possible RAW data races. Once execution reaches the point at which the log file is accessed (read) and determined to be closed, then the technique discovers that no synchronization was specified between the current point and the last point at which the log was actually closed (by a different thread). Thus, a RAW data race is detected at this point. Finally, it is determined that at the point at which the log is closed, another thread (the one needing to perform the update to the log) is also ready. Thus, the technique executes the program again and forces the update to the log to occur *before* the log is closed by the other thread. It turns out that this new

interleaving changes what is contained in the log, so it is determined that the RAW data race is potentially harmful, and it is reported as the root cause of the failure.

`Mysql-3`. In this program, there is a memory error in which the act of loading data from an input file into a database table can cause a segmentation fault, if no database has been first selected. The error in this case is an uninitialized read of a variable that should be associated with an open database, but instead unexpectedly contains the value `NULL`.

When the technique is run on this program, it is determined that a segmentation fault occurs at a call to `strlen` when the pointer passed to it is `NULL`. The technique determines that the last definition of this `NULL` pointer occurs at an earlier instruction instance in the execution. When the technique re-executes the program, control reaches the initial definition of the `NULL` variable. At this point, the technique determines that there are no prior accesses to this variable from other threads, so there are no WAR or WAW data races at this point. Thus, execution continues until it reaches the point of the second memory access (the read of the value `NULL`). It is determined that the write and subsequent read are from two different threads, but in this case, there exists specified synchronization between the two memory accesses, so no RAW data race occurs here. Thus, execution continues by suppressing all instruction instances directly or indirectly affected by this `NULL` value. It turns out that at the end of this execution, no additional failures occur. Thus, the definition of the `NULL` value is reported as the root cause of the failure. This can help developers to understand that the execution unexpectedly tried to dereference the value `NULL` that was defined at this point.

`Prozilla-1`. In this program, there is potential for a stack buffer overflow at an

unchecked call to `strncpy`; in this case, no check is made to ensure that the source string will actually fit into the allocated destination buffer.

When the technique is run on an input that triggers the stack overflow, a crash occurs at the return of a function called `parse_html_mirror_list`, due to corruption of the return address of the function call on the stack. The technique determines that the last definition of the memory location associated with this corrupted return address, is in fact from the unchecked call to `strncpy`. Upon re-execution, there are no data races identified since all instructions associated with the failure are from the same thread. However, as a result of the suppression conducted on this execution, it turns out that no additional failures occur during execution. The technique then reports the faulty call to `strncpy` as the root cause of the failure.

`Prozilla-2`. In this program, there is potential for a stack buffer overflow of a variable called `buffer`, which is declared to be of fixed maximum size on the stack. A call to `sprintf` using this buffer can potentially write more data into the buffer than what will fit into the allocated size.

When the technique is run on this program using an input that triggers the overflow, a segmentation fault first occurs when dereferencing a pointer that is declared on the stack. This is due to the pointer variable on the stack being corrupted by the stack overflow. It is determined that the last definition of this corrupted memory location occurred in the unchecked `sprintf`. When the technique re-executes the program and performs suppression, another crash occurs at the return from a function named `http_fetch_headers` (due to the function call return address being corrupted by the stack overflow). It is determined that the corrupted return address was also last defined in the unchecked `sprintf`. The

program is then executed again to conduct further suppression, but a third failure occurs at the return of a function called `get_http_info`. The corrupted return address in this case is again last defined at the unchecked `sprintf`. Finally, when the program is executed once more to conduct suppression, no additional failures occur. Thus, the error is identified to be the unchecked call to `sprintf`. In all of these suppression executions, no data races were found.

`Axel`. In this program, there is potential for a stack buffer overflow in an unchecked call to `sscanf`. In this case, the destination stack buffer is declared to be of fixed size: 256 bytes. Since the source string can be of arbitrary size in this case, the stack buffer may overflow.

When the technique is run on this program, a first segmentation fault occurs when dereferencing a pointer that is declared on the stack. The last definition of the corrupted memory location was in the unchecked call to `sscanf`. Upon the next suppression execution, a new crash occurs at the return of a function called `conn_info`, due to a corrupted function call return address. Again, the last definition of the corrupted location was in the unchecked call to `sscanf`. Upon the next suppression execution, no additional failures occur, and the unchecked call to `sscanf` is identified as the error. No data races are detected in these suppression executions.

Overview of results. Overall in the experiments, it was found that Execution Suppression could accurately identify the root cause of the failures in all of the benchmark programs. On the other hand, these programs happened to involve little propagation of corrupted memory. In general, some multithreading errors may involve significant propagation of corrupted memory, and in these cases, it is expected that the technique may continue to

be effective due to the iterative nature of the execution suppression technique. Also in the experiments, only 2 – 4 program executions were required by the technique to identify the error in each benchmark program. Thus, the total time required to run the technique is quite small in a debugging context. Each execution may perform tracing and monitoring in addition to execution suppression, but in the experiments, no single execution took more than a few seconds to complete.

6.4 Summary

In this chapter, a generalized version of the Execution Suppression state alteration technique was developed that can be effective at locating memory-related errors and potentially-harmful data race errors in multithreaded programs. The technique can be fully automated, and requires as input only a faulty program and an input causing an execution failure. The technique ensures that the effects of a multithreading error can be reproduced on multiple program re-executions. The technique accounts for data races involving RAW, WAR, and WAW dependencies. The technique also includes a method for dynamically altering the order of scheduled threads to check whether an identified data race is potentially harmful. An experimental evaluation using a set of 7 real bugs in large-scale multithreaded programs has shown that the technique can be very effective at precisely identifying the root causes of failures caused by multithreading and other memory-related errors. Further, relatively few state-altering program executions were required to locate the error in each benchmark, making the approach relatively efficient in a debugging context.

In the next chapter, issues regarding the implementation of suppression are described, for the basic technique and the generalized version for multithreaded programs.

Chapter 7

Suppression Implementation Issues

In the previous two chapters (Chapters 5 and 6), the Execution Suppression technique for locating memory errors in both single-threaded and multithreaded programs was developed. In this chapter, details and issues related to the implementation of suppression are discussed.

First, a general implementation for suppression is described at a conceptual level. Next, a software-only implementation is discussed that makes use of the Valgrind dynamic binary translation framework [55, 103]. Then, two types of hardware support are considered. The first approach takes advantage of hardware support already available in Itanium processors that was designed for deferred exception handling. The second, more hardware-intensive approach uses additional memory augmentation, similar to the hardware used in *dynamic information flow tracking* (DIFT) techniques [24, 134]. Finally, the overheads of the different software and hardware-based implementations are compared. All implementations analyze program executions at the binary instruction level.

7.1 General Implementation

Figure 7.1 shows the conceptual steps involved in any implementation of suppression. The first main step is to identify the instruction instance that is responsible for defining a memory location that directly causes a crash. To enable this, a global counter is maintained that represents a dynamic instruction count (variable *global_cnt*). Every register and memory word is associated with a count value, which represents the instruction that last defined it. Thus, for every instruction that defines a memory word or a register, *global_cnt* is incremented and stored in the counter associated with the defined location (line 1). To allow for identifying the instruction instance responsible for a crash, the instruction counts associated with the two sources are stored (lines 2 – 3). Thus, if a crash occurs at the current instruction, the technique can easily identify the instruction instance responsible for defining the memory location causing the crash.

Once the instruction instance directly causing the crash is identified, the technique re-executes the program and begins suppression starting at the appropriate instruction instance (determined by the condition at line 4). The fact that execution is in suppression mode is represented by a global *suppress* flag (the flag is set in line 5). In addition to setting this flag, the target of the current instruction is marked as *corrupt* to indicate that it is corrupt/infected (line 6). Once the *suppress* flag is set, each executed instruction from this point follows the suppression semantics if either source location is marked as *corrupt* (line 9), otherwise it follows the regular semantics (lines 11 – 12). If the suppression semantics are followed due to at least one of the sources being marked *corrupt*, the *corruption* bit is propagated by setting the *target* (defined memory location) of the instruction to be *corrupt*. If the regular semantics are followed, the instruction executes normally, and the

```
Global variables:
 - global_cnt: the current dynamic instruction count
 - suppress_cnt1, suppress_cnt2: potential suppression points in the event of a crash
 - suppress: flag that indicates if in suppression mode, initially false
Variables associated with every register and memory word:
 - cnt: global count value associated with the instruction last defining this
     register/memory word
 - corrupt: a boolean that is true if register/memory word is corrupt, false otherwise

Case:  target = src1 op src2       // target, src1, src2  can be register or memory word

1.  target.cnt = ++global_cnt     // update target to current instruction count value
2.  suppress_cnt1 = src1.cnt       // note suppression points in case program crashes here
3.  suppress_cnt2 = src2.cnt

4.  if (global_cnt == suppress_cnt1  or global_cnt == suppression_cnt2)
5.    suppress = true             // initiate suppression mode
6.    target.corrupt = true       // mark this first corrupted location
7.  if (suppress)
8.     if (src1.corrupt or src2.corrupt)
9.        target.corrupt = true   // suppression semantics
10.    else
11.       target = src1 op src2    // regular semantics
12.       target.corrupt = false
```

Figure 7.1: General implementation of suppression.

target (defined memory location) of the instruction is marked as *not corrupt* since it was computed using only non-corrupt values (this step is necessary in case the target location was previously marked as *corrupt* in the execution). Note that the description in Figure 7.1 shows the steps for all data transfer and arithmetic instructions. In a control transfer function, both branches are skipped if the predicate is marked *corrupt*; static analysis of the program is used to identify the next instruction to execute.

7.2 Software-Only Implementation

7.2.1 Basic Technique for Single-Threaded Programs

The basic technique is implemented fully in software based on the high-level system design shown in Figure 7.2. The design consists of three main components: the *Valgrind*

Figure 7.2: High-level system design for the software implementation of the basic technique. The bi-directional arrows represent interactions between the system components.

Core, the *Suppression Execution Tool*, and the *Memory Error Locator*.

Valgrind Core. The *Valgrind* infrastructure [55, 103] provides a synthetic CPU in software and allows for dynamic binary instrumentation of an executing program. Valgrind includes a set of tools that perform certain profiling and debugging tasks, but new tools can be added to the infrastructure to perform customized instrumentation tasks.

Suppression Execution Tool. The Suppression Execution Tool was created as a new tool for Valgrind. The tool takes as input an executable program with an associated input, and a set of (possibly empty) instruction instances whose effects should be suppressed during execution (called "suppression points"). The tool then performs the execution while simultaneously carrying out two tasks required by the technique: tracing and suppression. The tracing is required to see which memory locations are accessed and when. The suppression is required to nullify the effects of certain instructions during execution, when searching for the first point of memory corruption.

For tracing during a given execution, the tool records a trace of the memory locations accessed (loaded from and stored to) during the execution. To do this, the tool instruments each non-suppressed load and store instruction to record the current program counter, its associated instance number, the type of instruction (i.e., load or store), and the address of the accessed memory location. This information makes it possible to identify which accessed memory location directly caused a memory failure, and which instruction

instance last defined that memory location. The identified instruction instance can then be specified as one of the suppression points for the next execution, i.e., for the next invocation of the Suppression Execution Tool.

For suppression during a given execution, the Suppression Execution Tool performs "suppression information flow tracking" at all instructions, as well as "actual suppression" at the appropriate load and store instructions. To do the suppression information flow tracking, the implementation associates every memory location and register with a *shadow location* that contains information about whether or not the associated location needs to have its effects suppressed. Initially, all shadow locations are marked as "not suppressed." At an instruction, if at least one of the used memory locations or registers is marked as "suppressed," then any defined memory locations or registers are also marked as "suppressed." On the other hand, if none of the used locations are marked as suppressed, then any defined locations are marked as "not suppressed." Tracking this information during execution ensures that any instructions directly or indirectly influenced by a suppressed location can have their effects suppressed as well. Memory locations are initially marked as suppressed when they are used in an instruction instance that is specified as a suppression point.

Besides tracking suppression information, "actual suppression" is performed at memory load and store instructions. At a store instruction instance that uses a suppressed location, the effect of the store is suppressed by *not* writing to the destination location. The effect is as if the store never occurred and the destination location retains whatever value was originally contained there. Similarly, for a load instruction instance that uses a suppressed location, the load is suppressed by *not* reading from the source location. The

effect is as if the load never occurred and the destination register being loaded into retains whatever arbitrary data was originally contained there. This arbitrary data will never be used since anything dependent upon it will be suppressed as well.

There are a few special considerations to make when suppressing. If an infected location is used in a conditional check, then the entire conditional structure involving the infected location is suppressed, since it depends upon the infected value. For example, suppression would be performed for an entire "if/else" structure or an entire loop if the associated condition uses an infected location at some point during execution. Although this solution is not general, it is simple and worked well in the conducted experiments. In order to determine where a conditional structure ends, the implementation relies on static analysis of the program structure at the source code level; this information is mapped back to binary level as needed.

Special consideration must also be made for infection of the return address of a function call. This must be specially handled because one cannot avoid a function return or simply jump to an arbitrary address upon function return. Instead, profiling data is used from the current and other test case executions to cause the function to return to a known, valid address (particularly, the target return address used most frequently).

Finally, infected input to system calls must also be handled. To do this, the technique simply refrains from making system calls when they involve at least one infected input value. The same approach can be used to handle library calls if desired, although this is not currently implemented. It is observed that suppressing system and/or library calls when they involve at least one infected input value, is an approximation. A more precise approach would be to only suppress those instructions within the function calls that

actually depend upon the infected inputs. However, for system calls in particular, Valgrind cannot be used to instrument instructions within a system call itself, since Valgrind actually executes system calls on the real CPU. Thus, the approximation of omitting system calls entirely when they involve at least one infected input, was done because of implementation restrictions. However, this approximation was still able to lead to good results in the experiments.

Memory Error Locator. This is the main driver module for the technique that manages the suppression re-executions and identifies the suppression points. Given a faulty program and its associated input, this module first invokes the Suppression Execution Tool using an empty set of suppression points to record memory access tracing information from the test case execution. From this, a first suppression point(s) is identified and passed as input to a second invocation of the Suppression Execution Tool. If another program failure occurs, then another suppression point(s) is identified and another re-execution is performed. Eventually, no failure will occur and the latest identified suppression point(s) is reported.

7.2.2 Generalized Technique for Multithreaded Programs

The generalized technique for multithreaded programs is implemented fully in software based on the high-level system design shown in Figure 7.3. The Valgrind infrastructure is composed of three components: (1) the *Valgrind Core*, which is the core functionality already provided by Valgrind; (2) the *Failure Repeatability Component*, which is a component implemented within Valgrind that ensures a failure in a multithreaded execution can be repeated on subsequent executions; and (3) the *Suppression and Data Race Detection Tool*,

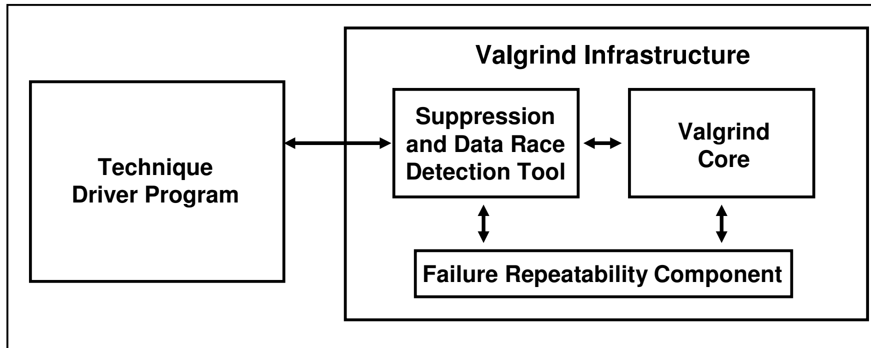

Figure 7.3: High-level system design for the software implementation of the generalized technique. Arrows represent interactions between the system components.

a tool implemented to carry out a program execution while performing the tasks necessary to conduct execution suppression as well as dynamic data race detection. Finally, a fourth component implemented outside of Valgrind called the *Technique Driver Program* works in conjunction with Valgrind to actually run the technique. Each of these components are now described in detail.

Valgrind Core. This is similar to what is described in Section 7.2.1, but here, the thread scheduler in the Valgrind core was customized to work in conjunction with the *Failure Repeatability Component* to ensure that the order in which threads are scheduled in a failing execution is repeated on subsequent executions. This ensures that the effects of an error can be repeated on multiple program executions, since the sequence of scheduled threads is preserved (for the experiments, it was not required to preserve other factors that could contribute to non-determinism, such as reading inputs from the execution environment).

Failure Repeatability Component. This component works in conjunction with the *Suppression and Data Race Detection Tool* and the *Valgrind Core* to ensure that the effects of an error can be repeated on multiple program executions. This component works as

follows. When the *Suppression and Data Race Detection Tool* carries out a failing execution for the first time, the *Failure Repeatability Component* records the order in which threads are scheduled during the execution. Then, for all subsequent executions conducted by the technique to isolate the same error, this component communicates with the *Valgrind Core* to guarantee that the threads are scheduled in the same order as in the original execution. This is accomplished by forcing the thread scheduler in Valgrind to schedule threads according to a specified schedule.

Suppression and Data Race Detection Tool. This tool was implemented within the Valgrind infrastructure to handle the two main tasks required for each program execution in the technique: (1) tracing and suppression; and (2) on-the-fly data race detection. The tool takes as input an executable program with associated input values for an execution, a set of instruction instances whose effects should be suppressed during execution (called "suppression points"), and a target write instruction instance (with associated memory location and value, and the ID of the thread performing the write) that must be checked to determine if it involves a potentially-harmful data race.

The tracing and suppression is performed in the same way as described in Section 7.2.1. For on-the-fly data race detection, the algorithm was previously shown in Section 6.1.2, Figure 6.2. To implement this, the technique requires as input a target write instruction instance i, with information about the thread t that performs the write, which memory location l is written, and the value v contained in the memory location after the write. During execution, all load and store (i.e., read and write) instruction instances executed prior to the target write instruction instance, are instrumented as follows: if the accessed location is l and the thread performing the access is *different than* t, then the

current instance is recorded as the most recent access to l from a thread other than t, and a global flag *sync* is set to $false$, indicating that no synchronization on accesses to l has yet been found. Later, if another instruction instance is executed that accesses l by a thread other than t, then all prior access information will be discarded and only information about this most recent access will be recorded. If any synchronization point related to l is encountered during execution, then the flag *sync* is set to *true*. When execution eventually reaches the target instruction instance i, then the technique knows which instruction instance last accessed the associated memory location from another thread, and whether any relevant synchronization was encountered between the two memory accesses. As a result, the technique can easily determine at this point whether a WAR or WAW data race is involved. Next, the technique sets the *sync* flag to $false$, and resumes execution at the instruction instance immediately following i. Again, if any synchronization related to l is encountered during execution, then *sync* is set to *true*. In the event that a load (read) occurs from location l by a thread other than t, and the *sync* flag is $false$, then a RAW data race has been found. When instrumenting the executing program in the implementation, it is assumed that synchronization points are clearly marked in the program so that they can be easily identified.

If any data race is identified in the implementation, then that race is checked to determine if it is potentially harmful. To accomplish this, the technique identifies the set of threads T that are ready, but are different than the thread originally involved, at the point of the first memory access in the data race. The set T can be identified on-the-fly when the first memory access occurs during execution. For each thread $x \in T$, the implementation re-executes the program and forces thread x to be scheduled in place of the

originally-executing thread at the point of the first memory access. The thread scheduler in Valgrind was customized to allow for control of the scheduler in this way. Then, after the instruction instance associated with the point of the second memory access is executed, the technique checks whether the value contained in the associated memory location has changed under this new thread interleaving. If so, then the data race is determined to be potentially harmful. In practice in the experiments, there were only a small number of threads available in each execution, so there were relatively few program re-executions required to identify the potentially-harmful data races.

Technique Driver Program. This is the main driver program for carrying out the generalized technique. Given a faulty program and an associated set of inputs that cause a failure, this program invokes the *Suppression and Data Race Detection Tool* using an empty set of suppression points and no specified target instruction instance at which to search for a data race. The *Suppression and Data Race Detection Tool* then records memory access tracing information from the execution, and simultaneously invokes the *Failure Repeatability Component* to record the order in which threads are scheduled during execution. Based on this information, an initial suppression point (and a target write instruction instance at which to search for a data race) is identified. This information is then passed to another invocation of the *Suppression and Data Race Detection Tool.* If no data race is reported but another failure occurs, then the process iterates again. Eventually, either a data race will be reported as the likely root cause, or else the execution will terminate without any failure, in which case the most recent suppression point is reported as the likely root cause.

7.3 Hardware Support

In this section, possible hardware support is described for reducing the overhead of suppression that would normally be required by a purely software implementation. First, it is shown how existing hardware support in Itanium processors (originally meant for deferred exception handling) can be leveraged to reduce the overhead of suppression. Then, it is shown how additional memory augmentation in the Itanium processor can further reduce the overhead of suppression.

7.3.1 Using Existing Support for Deferred Exception Handling

This implementation is motivated by the similarity of suppression to deferred exception handling that is performed in Itanium processors. Itanium processors allow instructions to be executed speculatively. However, speculative instructions can cause exceptions, and these exceptions should only be reported when any *non-speculative* instruction uses any speculatively-produced values. In other words, exceptions caused by speculative instructions are deferred and handled later. To implement this in the Itanium processor, whenever a speculative instruction causes an exception, the target (defined register) of the instruction is tagged with a special value, known as NaT (which stands for "not a thing"). These NaTs are propagated as the program executes, and when a non-speculative instruction uses any value tagged as NaT, an exception is finally reported. Itanium processors associate a NaT bit with every register, and the hardware automatically propagates NaT bits for register-based instructions.

It is observed that the *NaT bit* is analogous to the *corrupt bit* in the general implementation of suppression, and the propagation semantics of the NaT bit are identical

to the propagation semantics of the corrupt bit. Thus, one can take advantage of the NaT bits and the propagation hardware in Itanium processors when implementing execution suppression. The result is that lines 7 – 12 in the general implementation (Figure 7.1) are now automatically taken care of by the hardware, for register values. This results in significantly reduced overhead for the suppression technique. However, since there are no NaT bits associated with memory words in the Itanium processor, it is still required that the software handle the propagation of corrupt bits for memory instructions.

7.3.2 Additional Hardware Support through Memory Augmentation

In this technique, the Itanium processor is augmented with additional hardware support so that propagation of NaT bits is handled for memory instructions as well. To enable this, NaT bits are associated with every memory word. This results in the addition of a NaT bit for every memory word in main memory, as well as for every word in caches and in the external data bus. This further reduces the execution overhead. Figure 7.4 shows the architecture for the Itanium processor both with and without this additional memory augmentation.

7.4 Overhead Comparison

The overheads of suppression for each of the different implementations are now compared. Although the actual software implementation using the Valgrind infrastructure is targeted toward x86 machines, the hardware-based implementations needed to be implemented in a simulator. The SESC simulator targeting the MIPS instruction set was used. The simulations targeted an in-order processor with a 16-KB 4-way L1 cache and

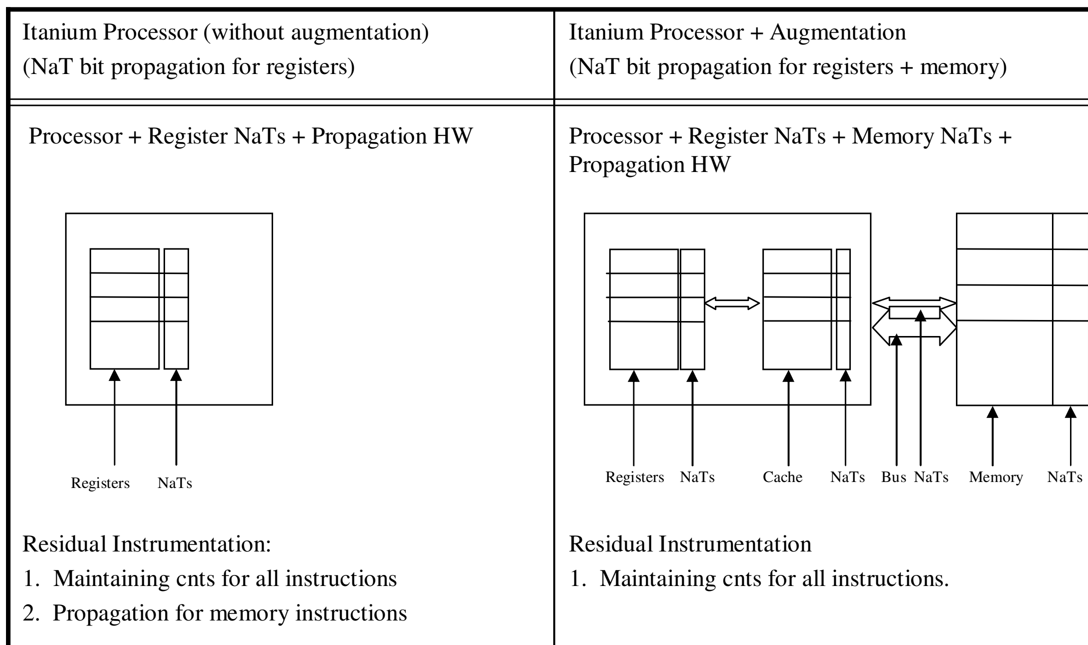

Figure 7.4: Itanium processor with and without memory augmentation.

a 512-KB 2-way L2 cache, with a memory latency of 250 cycles. Support for handling deferred exceptions was implemented in the simulator, in addition to implementing NaT bits for both register and memory locations. To obtain a fair overhead comparison between all implementations, the software implementation was run in the simulator as well, and the overhead was measured for only suppression (not including variable re-ordering or data race detection). The regular Valgrind-based software implementation incurs an overhead of around 50x – 100x, but this is because Valgrind is designed for ease of writing complex instrumentation tools, and not for optimizing performance. The software-based overhead is much lower when run in the simulator.

Figure 7.5 shows the overhead for all three implementations normalized to original execution time, for one run of each of the programs analyzed in the memory corruption

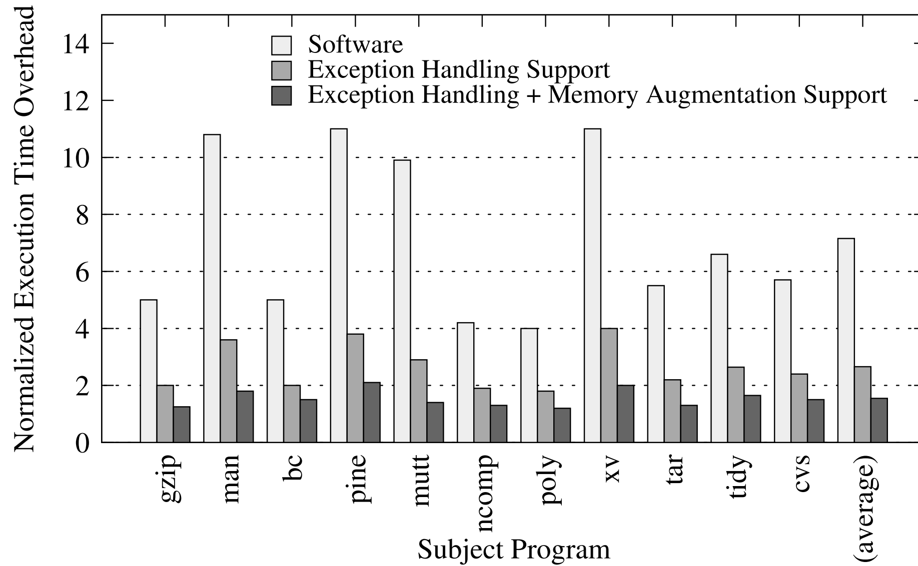

Figure 7.5: Execution time overheads for different suppression implementations.

study from Section 5.1. The right-most entry in the graph shows the average results. As shown in the figure, the software-only overhead is highest, with 7.2x overhead on average. Using the deferred exception hardware support available in the Itanium processor, this overhead can be significantly reduced to 2.7x on average. Finally, using additional memory augmentation support, this overhead can be further reduced to an average of 1.8x.

7.5 Summary

In this chapter, details regarding the implementation of suppression were discussed. Several different types of implementations were explored. First, a general implementation was presented at a conceptual level. Next, a software-only based implementation was described in detail. Then, an implementation was presented that takes advantage of hardware support for deferred exception handling that is available in Itanium processors. Finally,

a hardware-intensive scheme involving additional memory augmentation in the Itanium processor was considered. An experiment was conducted to compare the overheads of the various implementations for suppression. While the baseline software implementation incurred an average overhead of 7.2x, this overhead was reduced to 1.8x by using hardware support.

So far in this dissertation, two main dynamic state alteration techniques have been developed for automatically locating errors in software: Value Replacement and Execution Suppression. These techniques can be used to assist developers in more efficiently debugging faulty programs, because the techniques can help guide a developer to a faulty statement more quickly. However, in debugging, merely locating a faulty statement is only half the battle. The faulty statement must then be analyzed in the context of the program to determine an appropriate fix, so that the software can be modified to eliminate the error. Currently, there is relatively little research in the Software Engineering community that focuses on automated techniques to assist developers in fixing program errors, despite the fact that advances in this area can also significantly improve the efficiency of the debugging process. In the next chapter, a technique is developed to automatically assist developers in modifying suspicious statements to fix program errors.

Chapter 8

Towards Correction of Program Errors

In the previous chapters, techniques were developed that focus on locating faulty program statements. However, debugging a faulty program is not over once a faulty statement is found. The error in the statement must be eliminated by appropriately modifying the source code of the program. In some cases, the appropriate fix to make at a statement may not be obvious. It may require considerable time for a developer to manually analyze the debugging situation to understand why a statement is faulty, and then to figure out a way to modify the code to fix the error without introducing any new errors. Techniques to automatically assist developers in fixing program errors have potential to significantly improve the efficiency of the debugging process.

In this chapter, a technique is developed for automated assistance in fixing an error at a faulty program statement [66]. This technique is called *BugFix*. The goal of the technique is to automatically identify a set of relevant suggestions for modifying a

statement that is suspected of being faulty. A developer can consider these suggestions when determining an appropriate fix. To accomplish this, BugFix incorporates a machine learning-based algorithm that makes use of knowledge obtained from a history of previous faulty statements that were fixed, along with their corresponding *bug-fix descriptions*.

Definition 11 (Bug-Fix Description).

A **bug-fix description** *is a brief textual description of how to modify a particular faulty statement such that an error in the statement is fixed (e.g., “change the < operator into the <= operator”).*

Thus, the suggestions that are reported by the technique for fixing a new error, are a subset of the bug-fix descriptions currently known from the history of prior bug fixes. As a result, in order to be useful, the technique first requires an initial training phase in which a history of debugging situations and their corresponding bug-fix descriptions are learned.

BugFix requires as input a faulty program and a corresponding test suite containing at least one failing test case. The goal of BugFix is to compute and report a prioritized list of *bug-fix suggestions* for a given *debugging situation* at a program statement that is suspected of being faulty. A *debugging situation*, described in detail in Section 8.2.1, can be thought of as a characterization of the particular static and dynamic details of a suspicious statement being debugged. A *bug-fix suggestion* is simply a bug-fix description relevant to a current debugging situation, which is reported to a developer and can assist in fixing a faulty statement.

The technique is built upon concepts from the *machine learning* community, which allow the technique to learn about new debugging situations and their corresponding bug fixes that are encountered over time. Through continued use, the ability of the technique

to report highly relevant bug-fix suggestions for new debugging situations is expected to improve. This is accomplished by maintaining a *database of bug-fix scenarios* describing the different debugging situations and corresponding bug-fix descriptions previously encountered. From this database, a machine learning algorithm for learning association rules can be applied to automatically generate a *knowledgebase of rules*, mapping different (general and specific) debugging situations to corresponding bug-fix descriptions. Each rule is also associated with a confidence value indicating how likely the rule is to be correct (i.e., how likely a particular debugging situation should indeed map to the given bug-fix description).

Given a new debugging situation, BugFix automatically analyzes it in conjunction with the knowledgebase of rules to compute and report a prioritized list of relevant bug-fix suggestions. Once the appropriate fix is made by the developer, then the new information about the current debugging situation and corresponding bug fix is added to the database of bug-fix scenarios. This enables a revised set of rules to be computed that can lead to more effective results when the technique is used again in the future. BugFix has the potential to help a developer more quickly discover, apply, and verify the appropriate fix for an error.

8.1 Association Rule Learning

An important component of BugFix is the ability to analyze a history of debugging situations and their corresponding bug fixes, to identify a set of rules mapping different *combinations* of these debugging situations to their likely bug fixes. This is important because when a new debugging situation is encountered, it is unlikely to *precisely* match any debugging situation previously seen in the history. Nevertheless, BugFix must be able to analyze the history to figure out which bug-fix descriptions are *most relevant* to the current

debugging situation. Thus, a set of rules mapping different combinations of debugging situations to bug-fix descriptions can help. In this section, background information is provided on association rule learning to describe how BugFix accomplishes this.

In the machine learning community, *association rule learning* [6] is a popular method for discovering the relationship between variables in databases. It has been widely used in many diverse application areas, such as marketing, intrusion detection, genetic engineering, and (in the current work) software debugging.

Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of n attributes called *items*, and $T = \{t_1, t_2, \ldots, t_m\}$ be a set of m *transactions* comprising the *database*. Each transaction in T is a subset of the items in I (a set of items is commonly referred to as an *itemset*). *Association rules* are derived from these transactions in the database. An association rule is defined in the form $X \rightarrow Y$ where $X, Y \subseteq I$ and $X \cap Y = \emptyset$. X and Y are called the *antecedent* and the *consequent*, respectively. A rule intuitively means that if the items in set X are present/true, then it is probable that the items in set Y are also present/true. For example, an association rule in the supermarket domain could be $\{eggs, bread\} \rightarrow \{milk\}$, which implies that if a customer buys *eggs* and *bread*, then the customer probably buys *milk* as well.

The notion of *confidence* has been introduced to measure the significance of a rule. The confidence of a rule $X \rightarrow Y$ is defined as $supp(X \cup Y)/supp(X)$, where $supp(X)$ is the *support* of itemset X, which is equal to the fraction of transactions in the database containing X. This confidence can be interpreted as an estimate of the probability $P(Y|X)$. It allows one to select a subset of the most interesting rules from a set of all possible rules.

A variety of techniques have been developed for learning association rules. One of the most popular algorithms is `apriori` [7]. Given a database of transactions, `apriori`

identifies association rules through two key steps.

1. **Find the frequent itemsets**. These are sets of items for which the associated support values are at least a specified minimum value. The algorithm iteratively generates candidate itemsets and prunes out those containing subsets of items that are known to be infrequent.

2. **Generate association rules from the frequent itemsets**. For each frequent itemset X, `apriori` enumerates all non-empty subsets of X. For each such subset $Y \subseteq X$, the algorithm calculates the confidence of rule $Y \rightarrow (X - Y)$ and outputs the rule if the associated confidence value is larger than a specified minimum confidence.

BugFix uses the `apriori` algorithm to identify rules mapping debugging situations to bug-fix descriptions. These rules are then analyzed in conjunction with a newly-encountered debugging situation to identify the most relevant bug-fix suggestions from among the bug-fix descriptions currently known.

8.2 BugFix: Automated Assistance for Fixing Errors

BugFix assumes that an existing error location technique, for example, Value Replacement or Execution Suppression, is first used to locate a suspicious statement that is likely to be faulty. Once such a statement is found, then BugFix can be used to assist a developer in fixing an error at that statement. This is accomplished by performing the three main steps shown in Figure 8.1.

In the first step of the technique, the current *debugging situation* is analyzed and characterized in terms of both static (structure of the statement) and dynamic (patterns in

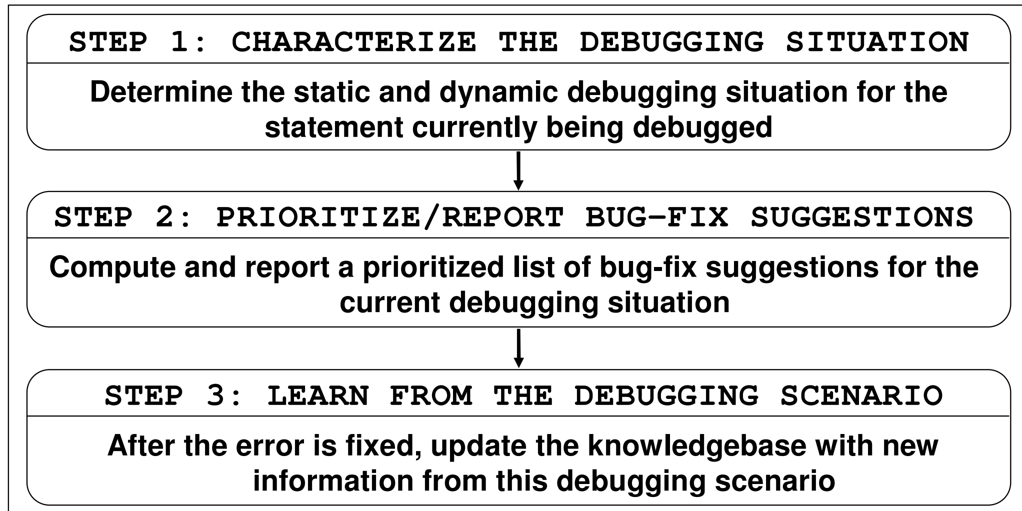

Figure 8.1: The three main steps of BugFix.

the values associated with the statement) information. In the second step, a *knowledgebase of rules* mapping various debugging situations to relevant bug-fix descriptions is queried. Based on the current analyzed debugging situation and the confidence values associated with each rule in the knowledgebase, a prioritized list of bug-fix suggestions relevant to the current debugging situation is computed and reported to the developer. In the third step, once the developer fixes the error in the statement, the knowledgebase of rules is updated with new information concerning the most recently-encountered debugging situation and the corresponding bug fix. Each of these three main steps are now described in detail.

8.2.1 Analyzing the Debugging Situation

Intuitively, a *debugging situation* is a characterization of a particular suspicious statement that is being examined during debugging. It can involve static details of the statement, such as the structure and context within the program. It can also involve dynamic details about the values used at the statement when exercised by test cases. BugFix

requires that a debugging situation be formally described so that algorithms can be used to analyze it. To do this, the notion of a *situation descriptor* is defined.

Definition 12 (Situation Descriptor).

A **situation descriptor** *is an atomic entity that describes a particular static or dynamic detail of a suspicious statement being debugged.*

Situation descriptors can be any atomic entities that describe what is going on at a considered suspicious statement. Given this definition, the notion of a *debugging situation* can be formally defined.

Definition 13 (Debugging Situation).

A **debugging situation** *is a set of situation descriptors associated with a suspicious statement that is being debugged.*

Intuitively, when two debugging situations are similar to each other, then they will have similar sets of situation descriptors. These situation descriptors represent a way to automatically characterize and compare different debugging situations to see how similar they are. Since situation descriptors can be either static or dynamic, three different types of situation descriptors are considered in the current work: (1) those associated with the static structure of the given statement; (2) those associated with patterns in the IVMPs associated with the statement (recall the definition of an IVMP from Section 2.1); and (3) those associated with patterns in the values used at the statement by failing and passing runs. Each of these types of situation descriptors are now described.

Statement Structure Situation Descriptors

The situation descriptors pertaining to statement structure are derived from the (unordered) *tokens* comprising the statement, as obtained by tokenizing the statement according to the programming language. To limit the total number of possible descriptors, some tokens are represented abstractly as general situation descriptors. For example, there are a theoretically-infinite number of different variable names and constant values, so these are represented by general descriptors such as "int-VAR" or "char-CONST". Other tokens such as keywords and operators come from a limited set of possibilities for the given language, so these are represented by situation descriptors named after the keywords/operators themselves. Finally, comments and formatting tokens such as semicolons, parentheses, and curly braces are ignored.

Figure 8.2 describes different `C` program structure entities and how they are treated as situation descriptors by the technique. From this figure, notice that for variable names and constant values, the technique actually associates two general situation descriptors for each: one without a type specifier, and one with a type specifier. This is because, for example, even if two different situations use constants of different type, then they should still be regarded as "slightly similar" because they both involve constant values. On the other hand, if both situations use constants of the same type, then they should be regarded as "very similar." The reference and dereference operators "&" and "*" need to be renamed as situation descriptors to avoid conflicts with the bitwise-and and multiplication operators, respectively. There are also general descriptors for assignment statements, conditional statements, and declaration statements. Since `C` allows for user-defined types, such types are specified as "user-defined" in situation descriptors that require a type to be specified.

<table>
<tr><th>C Structure Entity</th><th>Examples</th><th>How to Treat as Situation Descriptor</th><th>Situation Descriptor</th></tr>
<tr><td>keyword</td><td>if switch for case while default return</td><td>use as is</td><td>(the keyword itself)</td></tr>
<tr><td>operator</td><td>+ - * / && || & ~ <= ! -> .</td><td>use as is</td><td>(the operator itself)</td></tr>
<tr><td>ref / deref</td><td>& *</td><td>rename</td><td>REF DEREF</td></tr>
<tr><td>variable name</td><td>foo bar i j sum total average max</td><td>generalize</td><td>VAR x-VAR (x is the variable type)</td></tr>
<tr><td>constant value</td><td>17 “rabbit” ‘y’ 4 3.21 0 ‘z’ -3</td><td>generalize</td><td>CONST x-CONST (x is the constant type)</td></tr>
<tr><td>function call</td><td>foo() bar(x, 53) fprintf(“res: %d”, a)</td><td>generalize</td><td>FUNC_CALL</td></tr>
<tr><td>array access</td><td>x[10] foo[a+7] a[3]</td><td>generalize</td><td>ARRAY_ACCESS</td></tr>
<tr><td>cast</td><td>(int)a (char)(2+c)</td><td>generalize</td><td>CAST</td></tr>
<tr><td>format tokens</td><td>, { (;] :) [</td><td>ignore</td><td>(none)</td></tr>
</table>

Other general situation descriptors:

ASSIGN_STMT (for assignment statements)
COND_STMT (for conditional statements)
DECL_STMT (for declaration statements)

Figure 8.2: Deriving situation descriptors from various C program structure entities.

Figure 8.3 shows an example of some C statements and how they are represented by situation descriptors. The middle column shows how the C statement is tokenized into descriptors from left-to-right. The right-most column shows the final, unordered set of descriptors obtained by removing duplicate descriptors. In structures such as casts, function calls, and array references, the tokens contained *within* these structures are tokenized into descriptors as well.

IVMP Pattern Situation Descriptors

Whereas statement structure situation descriptors refer to static details about a suspicious statement, IVMP pattern situation descriptors refer to dynamic details about

C Statement	Tokenized/Converted into Situation Descriptors	Final Set of Descriptors
int x = a + b;	ASSIGN_STMT, int, VAR, int-VAR, =, VAR, int-VAR, +, VAR, int-VAR	ASSIGN_STMT, int, VAR, int-VAR, =, +
c = (char)(2 + *y);	ASSIGN_STMT, VAR, char-VAR, =, CAST, char, CONST, int-CONST, +, DEREF, VAR, int*-VAR	ASSIGN_STMT, VAR, char-VAR, =, CAST, char, CONST, int-CONST, +, DEREF, int*-VAR
if (foo(x) + a[3] < 0)	COND_STMT, if, FUNC_CALL, VAR, int-VAR, +, ARRAY_ACCESS, VAR, int*-VAR, CONST, int-CONST, <, CONST, int-CONST	COND_STMT, if, FUNC_CALL, VAR, int-VAR, +, ARRAY_ACCESS, int*-VAR, CONST, int-CONST, <

Figure 8.3: Example of C structure situation descriptors (assume type "int" when unspecified).

the statement when executed by test cases. IVMP pattern descriptors are derived from patterns that are observed in the IVMPs associated with the given statement. Recall that an IVMP, previously defined in Section 2.1, shows the original set of values used at an executed instance of this statement, and the corresponding set of alternate values that can be substituted at that point to cause a failing run to pass. Patterns that are observed in the values of these IVMPs can provide important clues about how a statement can be modified to fix an error. For example, if the error in a predicate statement is that the predicate is erroneously negated, then IVMPs at the statement are likely to indicate that the predicate outcome should be reversed, i.e., IVMPs will show a pattern in which all original values related to the predicate outcome will be negated in the alternate value sets. Since BugFix is a technique that provides suggestions for fixing errors, and IVMPs provide hints about how dynamic values at a statement should be changed to correct the output of failing runs,

then IVMP patterns can be very useful as situation descriptors.

BugFix uses available test cases to search for IVMPs at the given suspicious statement. Then, the IVMPs are analyzed for patterns that can be represented by situation descriptors. A *pattern* is considered to occur when corresponding values in the IVMPs compare to each other in the same way across all IVMPs at a statement. The IVMP example from Figure 2.1 in Chapter 2 involved a pattern in which the two used values are always the same in the original sets of values in the IVMPs. Another pattern could be when a particular original value always corresponds to a *larger* alternate value in the IVMPs.

To identify patterns in the IVMP values, consideration is made for how pairs of values compare to each other in terms of whether they are *less than*, *greater than*, or *equal to* each other. This is done by looking at pairs of values in three different ways: (1) within just the original sets of values in the IVMPs; (2) within just the alternate sets of values; and (3) between corresponding values in the original and alternate sets of values. For example, consider a statement with one defined value and two used values, so that in an IVMP the original set of values is $\{origDef, origUse1, origUse2\}$, and the alternate set of values is $\{altDef, altUse1, altUse2\}$. Then the original values are first compared to each other: $origDef/origUse1$, $origDef/origUse2$, $origUse1/origUse2$. Next, the same is done for the alternate values: $altDef/altUse1$, $altDef/altUse2$, $altUse1/altUse2$. Finally, the corresponding values between the original and alternate value sets are compared: $origDef/altDef$, $origUse1/altUse1$, $origUse2/altUse2$.

Figure 8.4 shows a more detailed example of how these pairs of values are compared, in the three columns labeled "Value Comparisons." These comparisons are computed for each IVMP associated with a statement (three IVMPs are shown in Figure 8.4). If cor-

IVMPs				Value Comparisons		
	Def	Use1	Use2	Original	Alternate	Corresponding
orig:	5	4	1	origDef > origUse1	altDef > altUse1	origDef < altDef
alt:	8	5	3	origDef > origUse2	altDef > altUse2	origUse1 < altUse1
				origUse1 > origUse2	altUse1 > altUse2	origUse2 < altUse2
orig:	10	3	7	origDef > origUse1	altDef > altUse1	origDef < altDef
alt:	11	10	1	origDef > origUse2	altDef > altUse2	origUse1 < altUse1
				origUse1 < origUse2	altUse1 > altUse2	origUse2 > altUse2
orig:	1	0	1	origDef > origUse1	altDef > altUse1	origDef < altDef
alt:	3	0	3	origDef = origUse2	altDef = altUse2	origUse1 = altUse1
				origUse1 < origUse2	altUse1 < altUse2	origUse2 < altUse2

Patterns (situation descriptors) found:

```
origDef > origUse1
altDef  > altUse1
origDef < altDef
```

Figure 8.4: Example of identifying patterns (situation descriptors) in IVMPs.

responding comparisons match across all IVMPs at the statement, then it is considered to be a pattern and is therefore designated as a situation descriptor (these are highlighted in Figure 8.4). General names are used to represent the IVMP values, such as $origDef$ or $altUse2$, so that the names in the descriptors do not vary among different statements or programs. Three additional patterns may also appear in IVMPs that are represented with "special" descriptors: descriptor *OTHER-BRANCH*, when a branch outcome always changes to the alternate outcome in the IVMPs; descriptor *ONE-TO-ANY*, when a single unique value in the original value sets always changes to some other value in the alternate value sets; and descriptor *ANY-TO-ONE*, when some original value always changes to a single unique alternate value. In the example in Figure 8.4, none of the "special" situation descriptors apply.

Value Pattern Situation Descriptors

These situation descriptors represent patterns in the values involved at a suspicious statement when exercised by both passing and failing runs. BugFix uses executions of the available test cases to identify the various sets of values used at the given statement, and these value sets are classified into two groups: those coming from failing runs, and those coming from passing runs. The technique searches for patterns among these values in a similar way as was done for the IVMP pattern situation descriptors. First, for each set of values exercised by failing runs, the technique sees how pairs of values compare to each other and then determines which comparisons are consistent across all failing-run value sets; the consistent relationships are designated as situation descriptors. Next, the same is done for the passing run value sets. Finally, a check is made for four additional patterns that are represented using the following "special" situation descriptors: descriptors *ALL-FAIL-SMALLER* and *ALL-FAIL-LARGER*, if a particular value from the failing run value sets is respectively always smaller or always larger than the corresponding value from the passing run value sets; and descriptors *ONE-FAIL-VALUE* and *ONE-PASS-VALUE*, if a particular value is the same in all failing run value sets or in all passing run value sets, respectively.

Figure 8.5 shows an example with three exercised value sets from failing runs, and four exercised value sets from passing runs. The comparisons between all pairs of values in the failing and passing value sets are shown. In this case, one value comparison pattern is consistent across all failing runs and is represented by a situation descriptor, and the special *ONE-FAIL-VALUE* situation descriptor applies as well because value $Use2$ in all the failing runs is 2.

Exercised Value Sets						Value Comparisons	
Failing Runs			Passing Runs			Failing Runs	Passing Runs
Def	Use1	Use2	Def	Use1	Use2		
0	0	**2**	1	4	4	failDef = failUse1 **failDef < failUse2** failUse1 < failUse2	passDef < passUse1 passDef < passUse2 passUse1 = passUse2
1	8	**2**	1	13	2	failDef < failUse1 **failDef < failUse2** failUse1 > failUse2	passDef < passUse1 passDef < passUse2 passUse1 > passUse2
1	7	**2**	0	0	0	failDef < failUse1 **failDef < failUse2** failUse1 > failUse2	passDef = passUse1 passDef = passUse2 passUse1 = passUse2
			0	7	5		passDef < passUse1 passDef < passUse2 passUse1 > passUse2

Patterns (situation descriptors) found: `failDef < failUse2`

Special patterns (situation descriptors) found: `ONE-FAIL-VALUE`

Figure 8.5: Example of identifying patterns (situation descriptors) in exercised value sets.

8.2.2 Prioritizing Bug-Fix Suggestions

Once the set of situation descriptors to characterize the current debugging situation has been determined, BugFix uses it to query a *knowledgebase of rules* that map various debugging situations to bug-fix descriptions. The result of this query is a prioritized list of bug-fix suggestions that is relevant to the current debugging situation. The knowledgebase of rules is first described, and then it is shown how the knowledgebase of rules can be used to compute a prioritized list of bug-fix suggestions relevant to the current debugging situation.

The Knowledgebase of Rules

The knowledgebase of rules is derived from a *database of bug-fix scenarios* that is maintained by the technique.

Definition 14 (Database of Bug-Fix Scenarios).

The **database of bug-fix scenarios** *is the history of previous debugging scenarios encountered in which a faulty statement was fixed. Each bug-fix scenario in the database is represented by a debugging situation and its associated set of bug-fix descriptions.*

This database is initially created through *training data* composed of some set of known debugging situations and their corresponding bug-fix descriptions. Each time BugFix encounters a new debugging situation and its corresponding set of bug-fix descriptions, this new scenario is added to the database. The *knowledgebase of rules* is then just the result of running the `apriori` association rule learning algorithm [7] on the database of bug-fix scenarios.

Definition 15 (knowledgebase of Rules).

The **knowledgebase of rules** *is a collection of rules, each with an associated confidence value, mapping various subsets of debugging situations to corresponding bug-fix descriptions. It is computed using the* `apriori` *association rule learning algorithm on the database of bug-fix scenarios.*

Whenever new information is added to the database of bug-fix scenarios, the database is passed as input to the `apriori` association rule learning algorithm to compute a revised knowledgebase of rules.

It is observed that a particular bug fix can be described in more general or more specific terms. For example, changing an operator from < into <= can be described generally as an "operator mutation", and more specifically as "change < into <=". To account for this, the technique allows a developer to describe a bug fix using multiple bug-fix de-

scriptions. This allows the technique to be more versatile in reporting the most relevant bug-fix suggestions for a debugging situation: sometimes, a more general bug-fix suggestion may be appropriate, while a more specific suggestion may be misleading.

The knowledgebase of rules is such that if a rule R exists that has a particular debugging situation S and bug-fix description F, then other rules will exist in which various subsets of S map to the same F. However, the confidence values associated with these other rules must be less than or equal to the confidence of rule R. For example, assume that a person who buys eggs and bread almost certainly also buys milk. Then if a person does indeed buy eggs and bread, one would have high confidence that the person will also buy milk. However, if another person buys only eggs, then that person may also buy milk, but one would have less confidence that this will be the case.

Figure 8.6 shows an example of what three rules might look like in the knowledgebase of rules for a conditional statement. In the figure, there is a single debugging situation composed of 9 situation descriptors that is mapped to three different bug-fix descriptions (from more general to more specific), each with a different confidence value.

Prioritizing the Bug-Fix Suggestions

Given a current debugging situation and a knowledgebase of rules, BugFix prioritizes the bug-fix descriptions in the knowledgebase of rules by performing four steps: (1) identifying rules to consider; (2) sorting rules by confidence value; (3) breaking ties by number of situation descriptors; and (4) reporting the prioritized bug-fix descriptions as suggestions.

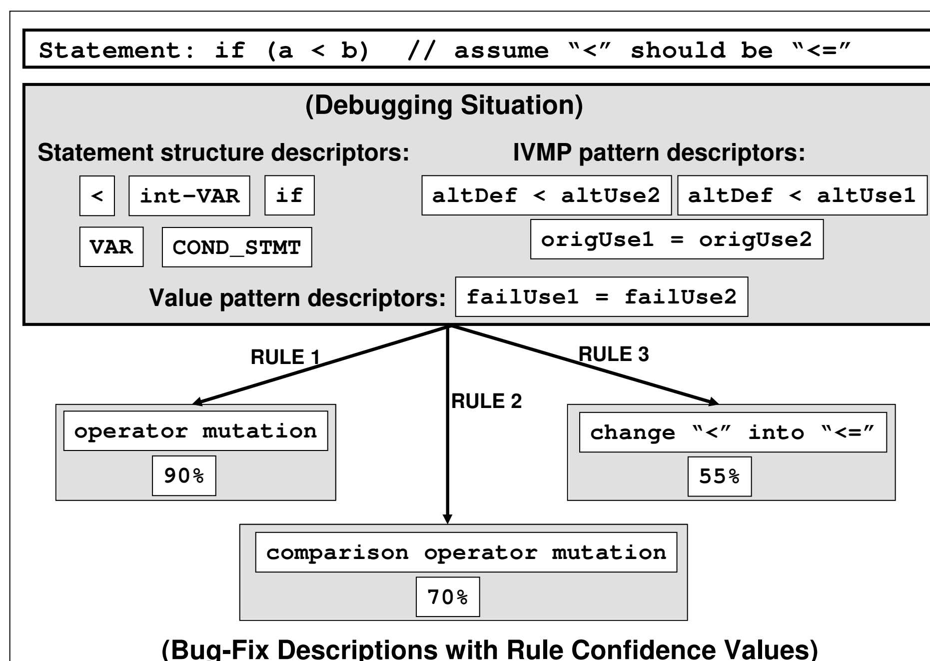

Figure 8.6: Example of three rules from a knowledgebase of rules (one debugging situation mapped to three bug-fix descriptions).

Identifying rules to consider. First, the subset of rules to be considered for prioritization is identified. These rules are those in which the debugging situation associated with the rule is a *subset* of the current debugging situation. Only these rules are considered because any rule that is *not* a subset will involve at least one situation descriptor that *does not apply* to the current debugging situation.

Figure 8.7 shows an abstract example in which there are 35 rules in the knowledgebase, and the bug-fix descriptions are ranked with respect to a current debugging situation. In the figure, each rule is shown with capital letters to represent situation descriptors, and lower-case letters to represent bug-fix descriptions. Confidence values are in parentheses after each rule. An asterisk (*) before a rule indicates that the rule is considered for prioritization since its set of situation descriptors is a subset of the current debugging situation.

Sort rules by confidence value. The rules being considered are ranked in

Current debugging situation: `A B D F`

RULE	PRIO-1	PRIO-2	RULE	PRIO-1	PRIO-2
* A -> a (33%)	4	8	* A D -> a (100%)	1	2
* A -> c (33%)	4	8	A E -> c (100%)		
* A -> e (34%)	3	6	B C -> a (100%)		
* B -> a (14%)	6	11	* B D -> a (71%)	2	3
* B -> b (12%)	7	12	* B D -> b (29%)	5	9
* B -> c (33%)	4	8	B E -> b (12%)		
* B -> d (7%)	8	13	B E -> c (33%)		
* B -> e (34%)	3	6	B E -> d (55%)		
C -> a (100%)			C D -> a (100%)		
* D -> a (71%)	2	4	D E -> b (100%)		
* D -> b (29%)	5	10	A B C -> a (100%)		
E -> b (12%)			* A B D -> a (100%)	1	1
E -> d (55%)			A B E -> c (100%)		
E -> c (33%)			A C D -> a (100%)		
* A B -> a (33%)	4	7	B C D -> a (100%)		
* A B -> c (33%)	4	7	B D E -> b (100%)		
* A B -> e (34%)	3	5	A B C D -> a (100%)		
A C -> a (100%)					

Prioritized list of bug-fix suggestions (w/o duplicates): `a e c b d`

Figure 8.7: Example of prioritizing bug-fix suggestions for a debugging situation.

decreasing order of confidence value. In Figure 8.7, the computed ranking is shown in the columns labeled "PRIO-1" (rank value 1 is the highest rank).

Break ties by number of situation descriptors. Any ties are broken by ordering rules in decreasing order of situation descriptor set size. The rationale for breaking ties in this way is the following: if two rules have the same confidence value, then the rule that has more situation descriptors in common with the specified debugging situation is likely to be associated with a more-relevant bug-fix description. In Figure 8.7, the ranking computed in this step is shown in the columns labeled "PRIO-2."

Report prioritized bug-fix descriptions as suggestions. Finally, the bug-fix

descriptions are reported as suggestions in order of their associated prioritized rules. If there are duplicate bug-fix descriptions, then only the first occurrence of each one in the sorted list is reported. In Figure 8.7, the final result consists of prioritized suggestions a, e, c, b, and d.

8.2.3 Learning from the Debugging Scenario

Once a developer fixes an error in the considered statement, the final step of the technique is to *learn* from this newly-encountered debugging situation and the corresponding bug fix. This is done by allowing the developer to describe the bug fix in terms of one or more bug-fix descriptions, and then adding a new entry representing the current debugging scenario to the database of bug-fix scenarios. The database is then passed as input to the `apriori` algorithm, which computes a revised knowledgebase of rules.

BugFix is designed so that it can become more effective over time at accurately predicting the most relevant bug fixes for debugging situations. It is fully automated except for the step of actually fixing an error at a faulty statement and specifying the bug fix in terms of bug-fix descriptions. The technique currently assumes that an error can be fixed by modifying a single source code statement.

8.3 Case Study

A case study is now presented that is designed to illustrate the use and potential benefit of BugFix. This study shows how the technique can be used to derive helpful bug-fix suggestions for several debugging situations (assuming that the faulty statement has first been located). The study uses an implementation of the `apriori` association

Program Name	**# Lines of Code**	**Program Description**	**Faulty Versions Used**
tcas	138	altitude separation	v6, v9, v20
totinfo	346	statistic computation	v16
sched	299	priority scheduler	v3
sched2	297	priority scheduler	v7
replace	516	pattern substituter	v1, v23

Table 8.1: Siemens benchmark programs used in the case study.

rule learning algorithm obtained from [56]. For the faulty programs to be debugged, a subset of the Siemens benchmark programs are used [64], described in Table 8.1. These faulty versions were selected because they can be easily and clearly described in detail, and because they highlight interesting aspects of the technique that show the potential benefit of BugFix. To enable identification of IVMP and value pattern situation descriptors for the debugging situations, a branch-coverage adequate test suite was associated with each faulty program consisting of at least 5 failing runs and 5 passing runs, selected from test case pools associated with each Siemens program.

8.3.1 Training Phase

BugFix is designed to become more effective over time at reporting the most relevant bug-fix suggestions for a given debugging situation. However, initially the technique must be *trained* using a set of known debugging situations and their corresponding bug fixes. This ensures that an initial knowledgebase of rules will exist. With more training, the technique is expected to perform more effectively on new debugging situations. To illustrate training in the case study, the following faulty programs and their known bug fixes were used: `tcas` v6, `replace` v1, `schedule` v3, and `totinfo` v16. For `tcas` v6, the full information (faulty statement, debugging situation, and bug-fix descriptions) is

shown in Figure 8.8. For the remaining training programs, only the faulty statements and corresponding bug-fix descriptions are shown in the figure. Notice in the figure that for `tcas` v6, the IVMP pattern situation descriptors "$origDef < altDef$" and "$origDef > altDef$" seem contradictory. This is because IVMPs are computed with respect to binary instructions in the implementation, and IVMP patterns are included from different binary instructions if they are associated with the same program statement.

To learn from the four debugging scenarios in Figure 8.8, four different *itemsets* are created – one for each of the four debugging scenarios – by taking the union of the debugging situation descriptors and the bug-fix descriptions. Then these four itemsets (that comprise the current database of bug-fix scenarios) are passed to the `apriori` algorithm [7] so that the knowledgebase of rules can be automatically derived. When invoking `apriori`, the algorithm is instructed to only report rules in which antecedents are comprised of only situation descriptors, and consequents are each comprised of a single bug-fix description. This ensures that all rules map debugging situations to bug-fix descriptions. The considered rules are not limited based on *support* value, but the considered rules are limited to those with *confidence* value at least 80%; this value was found to yield good results in the case study, based on the training data.

Table 8.2 shows the bug-fix descriptions involved in this case study. Each description has an abbreviation as specified in the middle column. The right-most column describes when the corresponding description is learned by the technique, either during initial training, or else through the remainder of this case study as new debugging situations are encountered.

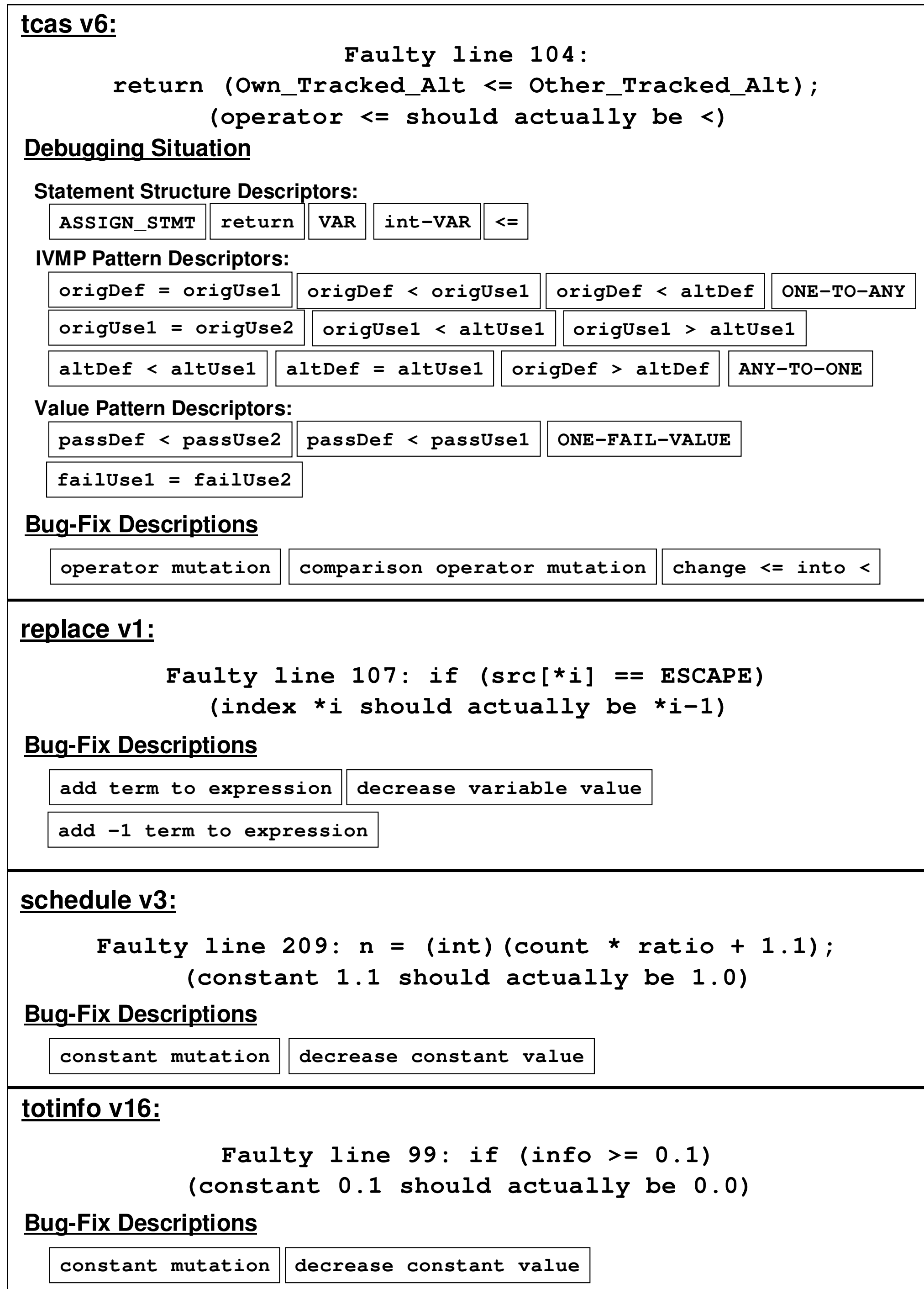

Figure 8.8: Four faulty program debugging scenarios used to train BugFix.

Bug-Fix Description	Abbreviation	When Learned
operator mutation	`opm`	training
comparison operator mutation	`copm`	training
change <= into <	`<=|<`	training
change >= into >	`>=|>`	new situation
add term to expression	`e+t`	training
decrease variable value	`v-`	training
increase variable value	`v+`	new situation
add -1 term to expression	`e-1`	training
add +1 term to expression	`e+1`	new situation
constant mutation	`c+-`	training
decrease constant value	`c-`	training

Table 8.2: Bug-fix descriptions involved in this case study.

8.3.2 Encountering New Debugging Situations

Based on the knowledgebase of rules obtained from the initial training, it can now be seen how the technique performs when encountering a new debugging situation. For this, four new debugging scenarios are considered, described in Figure 8.9 (the associated situations descriptors in the figure are omitted). Notice that `tcas` v9 and `tcas` v20 have faulty statements that look identical, but they are actually distinct statements occurring at two different source code lines in the program, so they are associated with distinct errors.

`Tcas v9`. This faulty program involves an error in which a comparison operator >= at line 90 should actually be >. For this debugging situation, the technique identifies the situation descriptors and then queries the (trained) knowledgebase of rules to obtain a prioritized list of relevant bug-fix suggestions. BugFix reports the prioritized list [`<=|<`, `copm`, `opm`, `e-1`, `v-`, `e+t`, `c-`, `c+-`], with associated rank values [1, 1, 1, 2, 2, 2, 3, 3]. In other words, the first three suggestions are tied for rank 1, the next three suggestions are tied for rank 2, and the last two suggestions are tied for rank 3. These results imply that potential bug fixes `<=|<`, `copm`, and `opm` should be considered first by a developer. Suggestions `copm`

```
tcas v9:
                    Faulty line 90:
upward_preferred = Inhibit_Biased_Climb() >= Down_Separation;
          (operator >= should actually be >)
```

```
tcas v20:
                    Faulty line 72:
upward_preferred = Inhibit_Biased_Climb() >= Down_Separation;
          (operator >= should actually be >)
```

```
replace v23:
          Faulty line 74: if (s[*i] == ENDSTR)
           (index *i should actually be *i+1)
```

```
schedule2 v7:
                    Faulty line 292:
    if (ratio < 0.0 || ratio >= 1.0) return (BADRATIO);
          (operator >= should actually be >)
```

Figure 8.9: Four new debugging scenarios (post-training) for the case study.

and `opm` are indeed effective, since the current situation does require a comparison operator mutation. Suggestion `<=|<` is less effective, but it is similar to the expected fix (the fix in this case, changing $>=$ into $>$, is not yet known to the technique). It is possible that these results can quickly guide a developer to an appropriate fix in this faulty statement. After the fix is made, suppose the developer describes the bug fix with three descriptors: *operator mutation*, *comparison operator mutation*, and *change* $>=$ *into* $>$. BugFix then learns from this current debugging scenario.

`Tcas` **v20**. This faulty statement looks identical to the one just seen in `tcas` v9, but since it is a distinct statement at a different source code line, it turns out that the debugging situation is slightly different due to some differences in the IVMP patterns. However, it is expected that the knowledge just learned from scenario `tcas` v9 should be beneficial in helping BugFix to report highly relevant bug-fix suggestions for this new situation. Indeed,

the prioritized bug-fix suggestions reported by the technique for this new situation are [`>=|>`, `copm`, `opm`, `<=|<`], with associated rank values [1, 1, 1, 2]. In this case, the other bug-fix descriptions contained in the knowledgebase of rules are not reported in the prioritized list because their associated rules all have confidence values less than the specified minimum threshold. All three suggestions with highest rank in the prioritized list precisely match the expected fix to make at this faulty statement. Thus, this demonstrates how the results reported by BugFix can improve over time as more knowledge is automatically learned through continued use of the technique.

`Replace` v23. In the faulty statement associated with this program, an array index $*i$ should actually be $*i + 1$. This error has some similarities to `replace` v1 that was involved during the training phase. The prioritized list of bug-fix suggestions in this case turns out to be [`e-1`, `v-`, `e+t`, `>=|>`, `copm`, `opm`, `c-`, `c+-`], with associated ranks [1, 1, 1, 2, 2, 2, 3, 3]. The highest-ranked suggestions (`e-1`, `v-`, and `e+t`) are, as might be expected, the same three bug-fix descriptions associated with the similar scenario `replace` v1 encountered during training. In this case, suggestion `e+t` is appropriate because a term should indeed be added to the array index expression. However, suggestions `e-1` and `v-` are not quite consistent with the expected fix, since here, the value of a variable should actually be *increased* by adding a +1 term to the index expression. On the other hand, these expected bug-fix suggestions (`v+` and `e+1`) are not yet known to the technique, and therefore could not have been reported. However, after the error is fixed, bug-fix suggestions `v+` and `e+1` are henceforth known to the technique, so more effective suggestions can be reported in a similar situation in the future.

`Schedule2` **`v7`**. Here, a debugging situation is encountered in a completely new subject program that has not yet been encountered in the study. Although the error in this case (comparison operator >= erroneously used instead of >) is familiar, the statement itself is rather unique as compared to what has been previously encountered. Based upon all knowledge learned from the previously-encountered debugging scenarios, the technique reports for the current scenario the following prioritized list of bug-fix suggestions: [`c-`, `c+-`, `>=|>`, `copm`, `opm`, `<=|<`, `e-1`, `v-`, `e+t`, `e+1`, `v+`], with associated ranks [1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 4]. In this case, the expected fix is represented by suggestions `>=|>`, `copm`, and `opm`, which are all given rank 2. However, note that rank-1 suggestion `c+-` also implies another possible fix: mutating the constant value 1.0. Indeed, instead of changing the predicate *ratio* >= 1.0 into *ratio* > 1.0 (the expected change), it may also be appropriate to instead mutate the constant so the predicate becomes *ratio* >= 1.001 (an unexpected change). It turns out that with the latter change, all available test cases pass. However, the latter change is not semantically equivalent to the former expected change. A developer must determine whether such a change is indeed appropriate, given the specification of the program.

Conclusions of case study. The case study illustrates that BugFix has potential to be effective at reporting relevant bug-fix suggestions for new debugging situations (given some amount of initial training), and that this effectiveness may improve over time due to the machine learning component that provides the foundation for BugFix.

8.4 Summary

In this chapter, a learning technique called BugFix is developed that can automatically assist developers in fixing program errors. The technique identifies a prioritized list

of bug-fix suggestions that are relevant to a given debugging situation. Through a machine learning algorithm, BugFix learns about new debugging situations and their corresponding bug fixes as they are encountered, thus increasing the effectiveness of the technique over time. A case study was also presented in this chapter, illustrating the use and potential benefit of BugFix.

The work on BugFix was originally motivated by the concept of IVMPs, previously described in Chapter 2. Since IVMPs show how values can be changed in an executed statement instance to cause the output of a failing run to become correct, IVMPs can provide important insights about how to correct program errors. However, in working on BugFix, it became apparent that considering IVMPs alone may not be sufficient; it is also important to consider other available information, such as the structure of a suspicious statement and the values exercised at the statement by passing and failing runs. By taking all of this information into account in a machine learning framework, the potential effectiveness of BugFix in reporting useful debugging suggestions is improved.

Chapter 9

Related Work

In this chapter, an overview of prior work is presented that is related to the work discussed in this dissertation. The focus of this dissertation is on automatically locating software errors using dynamic state alteration techniques. Prior work is first presented that focuses on locating software errors, starting with dynamic state alteration techniques. Then, prior work is discussed that focuses on fixing errors. Finally, prior work that focuses on tolerating errors is summarized.

9.1 Locating Errors

9.1.1 Techniques for Locating Errors in General

State Alteration Techniques

State alteration techniques for locating software errors, like the Value Replacement and Execution Suppression techniques presented in this dissertation, involve modifying the state of an executing program in order to isolate faulty statements. In the *Delta Debugging*

framework, a technique is described for isolating the differences between a passing and failing test case to simplify a failing test case into a minimal test case that still produces the failure [146]. This technique can then be applied to passing and failing executions to systematically narrow down the variables and values relevant to the failure, by modifying execution state and observing if there is any difference in the test outcome [145]. This can be used to identify the cause-effect chains relevant to a failure. Finally, focusing on *cause transitions* – moments in time when new variables relevant to a failure begin being failure causes – can help to isolate a faulty statement [22]. More recent work [95] has been proposed to significantly speed up Delta Debugging while increasing the quality of the results on tree-structured inputs. Delta Debugging is similar to Value Replacement because both techniques involve modifying execution states by swapping the values of variables in order to cause differences in test case outcomes. However, Value Replacement performs much more aggressive state alteration, potentially modifying the associated values at every executed statement instance in a failing run. Value Replacement also does not make use of passing runs except to gather profiling information about the values involved at each executed statement instance.

The idea of *Predicate Switching* [138, 148] involves altering the outcomes of particular predicates during execution of a failing run, to try to cause the run to become passing. This technique performs a subset of the state alterations that are performed in Value Replacement, because Value Replacement alters the outcome of predicates in addition to the values involved in any statement instance. Moreover, predicate switching may sometimes cause execution to enter inconsistent states, due to blindly altering only the control flow. Value Replacement, on the other hand, involves changing data flow, and a single value re-

placement can lead to multiple predicate switches in an execution that occur in a manner consistent with the revised values.

In general, state alteration techniques can potentially require significant time if many state alterations need to be performed in order to obtain useful information. Value Replacement was forced to deal with this issue, and so techniques were proposed in this dissertation to improve the efficiency of Value Replacement. Execution Suppression, on the other hand, performs more targeted state alterations, so time complexity of Execution Suppression is not as much of a concern as it is for Value Replacement.

Slicing Techniques

Slicing-based techniques identify subsets of program statements that can or do influence the value of a variable at some point in a program/execution. These techniques can be used to narrow down the set of statements that may be responsible for a failure, allowing a developer to locate an error more quickly. *Static slicing* [133, 139] identifies the subset of program statements that *may* affect the value of a variable at a particular program point, without making any assumptions about program input values.

Dynamic slicing [3, 4, 76, 78, 80, 107, 133] identifies the subset of program statements that *do* affect the value of a variable at a particular program point during a given execution. Dynamic slices are a subset of static slices. More recent work on dynamic slicing [147] has improved its effectiveness and efficiency for debugging. A series of precise dynamic slicing algorithms have been proposed [151], and a cost-effective dynamic slicing algorithm has been developed [150]. Work on compactly representing bytecode traces for slicing Java programs has also been conducted [137]. To help narrow down likely faulty

statements in dynamic slices, techniques have been proposed [82, 149, 152] to rank statements in the computed slices. Work has also been done on computing *dices* [5, 20] and *chops* [39, 41] to reduce the number of statements present in dynamic slices. Dices are computed by taking the set difference between slices, usually between the slices for correct and faulty variable values. Chops are computed by taking the set intersection between slices, usually between forward and backward slices. A recent survey of program slicing [142] includes discussion of some techniques for slicing multithreaded programs. A technique that was published after the survey discusses how slicing can be extended to capture data races in multithreaded programs [130].

To take potential dependencies into account, the notion of *relevant slicing* has been proposed [4, 79], and an algorithm has been developed for efficiently computing them [40]. Relevant slices are a subset of static slices but a superset of dynamic slices. In addition to those statements that actually influenced the value of a variable at a particular point during execution, relevant slices also contain statements that did *not* affect the variable value, but *could have* affected the value if a conditional statement had evaluated to a different outcome. Considering these additional statements can be useful in a debugging context.

Overall, slicing techniques consider static or dynamic control and data dependence information to identify subsets of program statements that may contain an error. This includes only *existing* state information; slicing does not consider altering dynamic state like what is discussed in this dissertation. Moreover, slicing reports a subset of program statements that can potentially be large. Unless techniques are used to rank statements in the computed slices, a developer may have to look through possibly many statements before an error can be found. In contrast to this, Value Replacement computes a ranked list of

program statements in which a faulty statement is very likely to be given high rank, and Execution Suppression reports only a single statement (or a few statements on rare occasions) that is very likely to be at or near to the location of an error. Execution Suppression also considers data dependence information when determining suppression points, but the technique may bypass possibly many links in the dependence chain between the error and the (original) failure, depending upon where crashes occur on each iteration of the technique. Slicing techniques, in general, would consider all links in all dependence chains influencing the value of a variable at a particular program point.

Statistical Techniques

Statistical techniques use dynamic information taken from some number of program executions in order to isolate program errors. In many cases, these techniques can be used to compute suspiciousness values for each program statement in order to rank the statements according to their likelihood of being faulty. These rankings can help guide a developer to an error more quickly than by blindly searching through all program statements.

Work has been done on analyzing and evaluating program spectra for the purpose of locating software errors [44]. It was discovered [43] that failing executions tend to have unusual coverage spectra as compared to passing executions. Inspired by this discovery, Jones et. al. developed a technique called *Tarantula* [74, 75] that ranks program statements according to likelihood of being faulty. The key idea of their technique is that statements exercised more often by failing runs than by passing runs, are more likely to be faulty. The *Nearest Neighbor* technique [119] takes a failing run along with a larger number of passing

runs, identifies the passing run that is the most similar to the failing run, and compares the spectra between these two runs to identify the most suspicious parts of a program.

Liblit et. al. have conducted work on *Cooperative Bug Isolation* [85, 86, 87], a series of techniques for locating software errors based on statistical analysis of sparse feedback data. This data is taken from program executions experienced by large numbers of software end users. Xie and Engler noticed [141] that redundant operations in programs are strongly correlated with the presence of errors such as null pointer dereferences. The *SOBER* technique [88] models the evaluation patterns of predicates in passing and failing executions, and considers a predicate to be relevant to an error if its evaluation pattern in failing executions differs significantly from that in passing executions. The *Artemis* technique [30] provides practical runtime monitoring of executions for error detection; the key idea is to only monitor those statement instances that are likely to be erroneous. To determine the likely erroneous statements, the technique builds a model of correct behavior through training from a set of passing runs, and identifies execution contexts that differ significantly from the model. Jiang and Su realized [72] that prior statistical techniques that identify error-predicting predicates may be ineffective for locating errors that are not directly associated with these predicates. They propose a technique to automatically generate faulty control-flow paths that link many error-predicting predicates together, which can be more effective at locating errors.

Unlike the state alteration techniques proposed in this dissertation, statistical techniques only consider existing state information about program executions. Value Replacement is similar to statistical techniques in that statements are ranked according to suspiciousness values. Both Value Replacement and other statistical techniques can lead to

false positives, statements with high suspiciousness that are actually not faulty and may be unrelated to an error. However, Execution Suppression will not lead to false positives in this sense, because any statement reported by Execution Suppression will be contained in the dependence chain between the error and the point of a resulting failure, though the reported statement is hopefully at or very close to the faulty statement.

Static Techniques

While all of the techniques discussed in this dissertation are dynamic techniques that involve analyzing actual program executions, there has also been significant work on static techniques for locating errors.

The *LCLint* checking tool [29] assumes annotations are written in a program to explicitly describe the results of functions and the values of parameters and global variables; these annotations can then be exploited by the tool to detect errors at compile time. The *PREfix* tool [17] performs compile-time analysis of a program to symbolically execute functions while modeling memory and identifying any observed inconsistencies that may be related to an error. Jackson and Vaziri [65] developed a technique in which a procedure is translated into a relational formula, which is then joined with the *negation* of the procedure's specification. A constraint solver is then used to search for potential executions of the code that can violate the procedure specification; these are likely to be associated with errors. The *SLAM* toolkit [11] creates abstractions of `C` code using iterative refinement, based on a specified temporal safety property of interest; the property is then automatically validated using the tool. *BLAST* [48] also verifies safety properties of `C` programs. *CQual* [34, 35] assumes that programs written in languages with static type systems are

annotated with a few type qualifiers; automatic type qualifier inference can be used to determine the remaining qualifiers, and then the system can check the annotations for consistency. The *PSE* technique [93] takes as input the type and source code location of an execution failure, and tracks the relevant value of interest back from the point of the failure through the points in the program where that value may have originated. Xie and Aiken present an error-detection tool [140] that exploits advances in boolean satisfiability solvers to translate programs into boolean formulas that can be solved to check for violations that can be correlated with errors. The *Extended Static Checker for Java* [33] looks for common programming errors at compile-time by way of an annotation language in which a developer formally expresses design decisions.

It has been observed that static techniques for locating errors often require programmers to modify software with annotations or specifications, and this can potentially place significant burden on developers. To address this issue, a body of research has been conducted [10, 27, 84, 89] on using data mining techniques to automatically infer specifications from software.

Other Techniques for Locating General Errors

Invariant-based techniques formulate invariants regarding the proper behavior of software; any violations to these invariants can then be examined as possible errors. The *Daikon* tool [28] can be used to automatically infer likely program invariants by dynamically analyzing program executions; these invariants can then be applied to debugging to discover potential errors when these invariants are violated at runtime. *DIDUCE* [42] can similarly be used to hypothesize likely invariants; the strictest invariants are hypothesized at first,

but as violations are detected and examined for possible errors, the hypothesis is relaxed over time to allow for newly-encountered program behavior. In *AccMon* [155], invariants are identified regarding the set of program instructions that typically access particular memory locations; when outlier instructions access these memory locations, there is a chance these instructions are associated with an error, perhaps an error that causes memory corruption such as a buffer overflow.

Check 'n' Crash [23] derives error conditions in a program statically and then generates concrete test cases to dynamically verify whether an error truly exists. *Eclat* [106] infers an operational model of the correct behavior of a program and identifies inputs whose operational execution patterns differ from the model in particular ways; these inputs are likely to be error-revealing. The `FindBugs` tool [49] automatically detects error patterns in Java programs. *PathExpander* [91] provides support to increase the path coverage of dynamic error-detection tools by executing non-taken paths in a sandbox environment. This allows for error detection in paths that would have otherwise not been analyzed.

A few techniques have been developed that focus on locating multiple simultaneous errors in software. Abreu et. al. [2] describe a dynamic model-based approach that can derive explanations for multiple potential errors in software. Jones et. al. [73] describe a framework for *parallel debugging*, in which failing runs are clustered according to a clustering technique, and then used to create specialized test suites that are each targeted to a single error. This enables the use of parallel work flows to debug different errors simultaneously, thereby reducing the time-to-release of a program. This dissertation has shown how Value Replacement can be generalized to also be effective in the presence of multiple simultaneous faults. However, whereas the work in [73] allows for parallel debugging, Value Replacement

is iterative in nature, focusing on effectively locating and fixing each error, one at a time.

To some degree, many techniques for locating general errors can also help to explain how those errors are revealed during execution. For example, in slicing techniques, the dependence chains traversed during computation of slices essentially express the cause-effect relations between an error and the resulting failure. Also, with the Value Replacement technique proposed in this dissertation, the computed IVMPs give hints about why values may have been wrong during a failing execution. In cases such as this, the error explanations obtained are products of the approach taken in order to compute the desired results; the explanations are a useful consequence of the technique, but do not represent the end goal of the technique (the end goal in these cases is to locate an error). There are a few existing techniques, however, that focus explicitly on trying to explain why a failure occurs due to an error. The work of Groce et. al. [37, 38], similar to the Nearest Neighbor technique [119], uses distance metrics to compare passing and failing program executions to isolate the differences, and then uses these differences to shed light on why an error is causing a failure. Ko and Myers [77] developed a debugging tool called *The Whyline* to help developers better understand program behavior. This tool allows developers to select a question concerning the output of a program, and the tool then uses a combination of static and dynamic analysis techniques to search for possible explanations.

9.1.2 Techniques for Locating Multithreading Errors

Detecting Data Races

Data races represent an important class of errors that is unique to multithreaded programs. As a result, there are a variety of existing techniques for identifying data races.

These techniques can be classified into those that use the *happens-before* algorithm [21, 94, 121], the *lockset* algorithm [31, 81, 124, 156], or a hybrid algorithm that combines both approaches [25, 105, 144]. The key idea behind lockset-based algorithms is to check whether shared variables are protected by at least one lock. The algorithms employ heuristics that can cause false positives to be reported. On the other hand, the key idea behind happens-before algorithms is to check whether accesses to shared variables in a program are explicitly ordered through synchronization operations. These approaches may miss some data races, but all identified data races will be true data races (though some of them may be benign). Hybrid algorithms generally attempt to achieve coverage close to that of lockset-based algorithms, while reducing false positives.

The *ReEnact* technique [112] dynamically detects data races in multithreaded programs by reusing architectural support for thread-level speculation. Work by Tallam et. al. [130] has shown how to extend dynamic slicing to consider additional types of dependencies for effective capturing of data races. This dissertation has shown how Execution Suppression can be extended to work for multithreaded programs and to handle data race errors. The proposed technique is based on happens-before relationships because it identifies conflicting memory accesses for which there is no explicit synchronization between them. Thus, all identified data races in the technique are true data races. Moreover, the technique further checks to see if a data race is potentially harmful [101] before reporting it as a likely root cause of a failure.

Static Techniques for Multithreaded Programs

Several techniques have been developed that focus on static checking of multi-threaded programs, to address the unique challenges in locating errors posed by multi-threaded code. Qadeer and Rehof observed [113] that although the problem of verifying a concurrent boolean program is undecidable, the problem can be made decidable if analysis is restricted to executions in which the number of context switches is bounded by a constant; such analyses can still discover intricate errors in software. The work was later refined by Musuvathi and Qadeer [97] in a technique for iterative context-bounding, in which the possible executions of a multithreaded program are systematically explored in an order that gives priority to executions with fewer context switches. *Calvin* [32] uses automatic theorem proving to statically check multithreaded programs in a scalable manner, requiring only a moderate amount of annotation overhead.

9.1.3 Techniques for Locating Specific Kinds of Errors

There have been a variety of techniques developed that focus on locating only specific types of errors.

Valgrind [103] and *Purify* [45] can be used to detect memory errors, but are restrictive in that they look for particular kinds of memory errors, such as buffer overflows and memory leaks. In one sense, the Execution Suppression technique proposed in this dissertation is more general because it can be used to locate any errors involving corrupted memory. On the other hand, *Valgrind* and *Purify* can detect some errors that may not involve memory corruption, such as memory leaks. *CCured* [102] is a technique for verifying type-safety of pointers both statically and during runtime, which can be used to find poten-

tial memory errors. However, the technique requires modifications to program source code. *HeapMon* [125] takes advantage of extra cores in hardware to improve the efficiency of error monitoring for heap memory errors. *SafeMem* [114] exploits ECC memory technology to detect memory leaks and memory corruption. Execution Suppression, on the other hand, identifies memory corruption through memory-related program failures.

There has been a variety of work on techniques for detecting buffer overflows, a particular type of memory error. For example, *Write Integrity Testing* (WIT) [8] uses a combination of static analysis and runtime instrumentation to ensure that instructions do not write to unintended storage locations, and control does not transfer to unintended targets; the average space and runtime overhead of their approach is around 10%. Ruwase and Lam's *CRED* tool [122] performs bounds-checking in order to detect buffer overflow attacks, incurring an overhead of 1% to 130%. While these techniques incur relatively low overhead, the main difference compared to Execution Suppression is that the bounds-checking approaches are designed specifically to capture out-of-bounds memory accesses. On the other hand, buffer overflows are only one type of memory error that can be located using Execution Suppression. Other errors that may not involve out-of-bounds memory accesses, such as double frees, uninitialized reads, and dangling pointers, can also be located by Execution Suppression.

The notion of *pointer taintedness* [19] can be used to detect security attacks that can be associated with a security error. The idea is to treat pointers as tainted if user input can be used to compute the pointer value; whenever a tainted value is dereferenced during execution, a security attack is detected.

CP-Miner [83] is a technique that searches for copy-paste errors in large-scale

software systems. *EXPLODE* [143] focuses on identifying data integrity errors in storage systems.

9.2 Fixing Errors

The techniques described in the previous section for locating errors sometimes produce accompanying information that can help developers to understand why the errors are causing failures. For instance, Delta Debugging can be used to identify cause transitions during execution when particular variables stop and start becoming causes for a failure. These cause transitions can help to explain what is going on between the point of error traversal and the point of a failure. In general, the information that can help in understanding program failures can also be used to assist in fixing those errors. However, the techniques described in the previous section have the primary goal of locating errors, not on modifying program code to eliminate errors. Compared to the amount of prior work on automated error location, there is relatively little prior work that focuses explicitly on automating the process of assisting developers in modifying program code to eliminate errors. Advances in both directions can help to improve the efficiency of the debugging process.

He and Gupta [46, 47] developed a technique to automatically generate program modifications to correct an erroneous statement in a function. In order to locate a likely erroneous statement, their technique combines ideas from software testing and path-based weakest preconditions, which are used in correctness proof methods. Their approach requires that a (correct) formal precondition and postcondition be specified for a function in terms of first-order theory formulas. Similar to this work, the BugFix technique described in this dissertation can automatically assist developers in modifying a likely erro-

neous statement to fix an error. However, unlike the technique of He and Gupta, BugFix does not require any formal preconditions or postconditions. The technique of He and Gupta automatically locates a likely erroneous statement, whereas BugFix assumes that such a statement has already been located using an existing error location technique. Also, the technique of He and Gupta can directly generate program modifications, whereas the BugFix technique can only report suggestions for how to modify program code taken from a prior, finite list of known suggestions.

Abraham and Erwig [1] developed a debugging tool for spreadsheets in which a user can specify the expected value for a cell that contains an incorrect value; the tool then identifies change suggestions that can be used to correct the error. In contrast to this, BugFix is not designed to work on spreadsheets, but on general program source code.

9.3 Tolerating Errors

If a program error is not being located and fixed, then other techniques must be used to deal with the negative effects of the error. This can be accomplished either by taking preventive measures to avoid the negative effects of the error in the first place, or else to deal with the negative effects of the error after they occur during execution (i.e., to recover from the effects of the error). Unlike the techniques described in this section for tolerating errors, the techniques proposed in this dissertation focus on locating errors so that they can be eliminated.

9.3.1 Avoiding the Negative Effects of Errors

The Ph.D. dissertation of Michael Bond [14] describes two techniques for avoiding memory leak errors. The first technique, *Melt* [15], identifies stale objects that a program is not accessing, stores these stale objects to disk, and activates these objects only if a program subsequently accesses them. The second technique, *leak pruning* [16], predicts leaked objects based on data structure usage patterns, and then reclaims these objects at runtime; an error is thrown if any reclaimed object is later accessed.

There has been recent work on protecting against heap-based memory errors to improve program reliability. *DieHard* [13] provides memory safety with high probability by randomizing the location of objects in a large heap and by replicating execution. *Archipelago* [92] allocates heap objects far apart in virtual address space to combat buffer overflows, and protects against dangling pointer errors by preserving freed objects after they are freed. *Exterminator* [104] pinpoints heap-based memory errors and derives runtime patches to avoid them in the current and subsequent executions.

Work on *Failure-Oblivious Computing* [120] allows servers to execute through memory errors without memory corruption. This is accomplished by using a safe compiler to insert checks into `C` programs that dynamically detect invalid memory accesses. When an invalid access occurs, invalid writes are simply discarded and values are manufactured to be returned for invalid reads. This allows a server to continue on its normal execution path without failing.

The *Samurai* system [108] provides safeguards against corruption of critical data through a memory model called *critical memory*. Their system uses replication and forward error correction to ensure that non-critical updates do not corrupt critical data. However,

the system requires that critical memory be explicitly identified by a programmer.

The idea of *data diversity* has been proposed [9] to avoid certain failures. Data diversity observes that when a program fails due to particular input, in some cases the program failure can be avoided if the input is changed in some minor way (according to the program specification). Further, if multiple copies of the same program are executed in parallel on slightly-different input sets, then a voting scheme can be applied to select the program output that is most likely to be correct.

9.3.2 Recovering from the Negative Effects of Errors

Once a failure occurs, the simplest way to recover from a program failure is to restart the whole program [36]. However, a more recent work [18] proposes selective restarting of a small set of partial software components, in order to reduce the cost of recovery. Both techniques, however, cause a program to be temporarily unavailable while all or part of the program is being restarted.

To further reduce the cost of recovery and avoid restarting a program, a variety of techniques incorporating the notion of checkpointing and logging have been proposed [36, 111, 115, 116, 127, 132, 153]. These techniques involve periodically checkpointing the state of an executing program. Once a failure occurs, the system can rollback to the most recent safe checkpoint and then resume execution from that point. Steps can be taken to avoid the failure the second time, such as by dropping a faulty user request in a server program.

Sidiroglou et. al. describe a framework for reacting to a wide variety of software failures [126]. Their system monitors the execution of a program for observed failures. In future executions of the program, the faulty regions of code are executed in an instruction-

level emulator; this emulator checks for recurrences of the conditions for previously-seen failures prior to the execution of each instruction. If such a recurrence is detected, then the program execution is recovered to a safe control flow.

Recovery-Oriented Computing [109, 110] is a framework that assumes that computer errors are inevitable, and therefore techniques to recover from those errors are important. The framework advocates for isolation and redundancy in system design, system-wide support for undo operations, integrated diagnostic support for efficient recovery from failures, and online verification of recovery schemes.

Work has also been done on recovering from failing device drivers [128, 129] and recovering from transient soft errors [96, 117, 118, 135, 136]; transient soft errors are radiation-induced errors that cause random bit-flips in hardware.

Chapter 10

Conclusions

10.1 Contributions of this Dissertation

The main contributions of this dissertation are in the area of dynamic state alteration techniques for locating software errors. Two major dynamic state alteration techniques were developed: Value Replacement and Execution Suppression.

Value Replacement performs aggressive state alteration by replacing the set of values involved in each statement instance in a failing execution with alternate sets of values. If any value replacement causes the output of the execution to change and become correct, then the statement associated with the value replacement is likely to be associated with an error. It was shown how Value Replacement can be generalized into an iterative technique for locating multiple simultaneous errors in software. Also, several techniques to improve the efficiency of Value Replacement were presented.

Execution Suppression performs targeted state alteration by iteratively identifying and avoiding the effects of statements in a failing execution that are known to involve

memory corruption. This process iterates until the first point of memory corruption in the execution is identified; this point is likely to be at or near to a memory error. It was shown how Execution Suppression can be extended to work effectively in the context of multithreaded programs and data race errors. Implementation issues of suppression were also considered, including some forms of hardware support.

A technique called BugFix was presented that makes use of the *interesting value mapping pairs* (IVMPs) computed by Value Replacement. BugFix is an automated, learning-based technique that provides relevant suggestions for how to modify a given suspicious statement to correct an error in that statement. It is one of only a handful of known techniques for providing automated assistance in modifying program code to eliminate errors.

The following research questions are addressed in this dissertation.

(Effectiveness) Can dynamic state alteration techniques effectively locate software errors?

Prior to the work conducted in this dissertation, the potential effectiveness of dynamic state alteration techniques in locating software errors was questionable. For example, when the work on *cause transitions* in the state-altering Delta Debugging framework was published [22], it was shown that cause transitions could achieve more effective error location results on the Siemens benchmark programs [64] than Nearest Neighbor [119], the most successful technique known up until that point. However, it was later shown [74] that the much simpler Tarantula statistical (non-state altering) technique could produce superior error location results on the same benchmark programs.

In this dissertation, it was shown that Value Replacement, which performs much

more aggressive state alteration than Delta Debugging, can achieve error location results that are significantly more effective than the results that can be achieved by Tarantula on the Siemens benchmark programs [64]. Although the basic Value Replacement technique can have reduced effectiveness in the presence of multiple simultaneous errors, this dissertation has also shown how to generalize Value Replacement into an iterative technique that can effectively locate each of the multiple errors.

The Execution Suppression technique described in this dissertation can accurately pinpoint the first point of memory corruption in an execution that fails due to a memory error; the first point of memory corruption is typically at or very close to the point of the error itself. Thus, Execution Suppression can be highly effective at locating memory errors. This dissertation has also shown how to extend Execution Suppression to effectively locate memory errors such as data races in multithreaded programs.

(Efficiency) Can dynamic state alteration techniques efficiently locate software errors?

Previously, it was not clear whether dynamic state alteration techniques could produce error location results that were effective enough to justify the time required to perform the techniques. For instance, cause transitions were shown [22] to require time ranging from several minutes to a few hours for each analyzed benchmark program. As another example, a full, brute-force implementation of Value Replacement can potentially require dozens of hours of computation time for each benchmark program considered in the experiments.

This dissertation described several ideas that can be used to drastically improve the

efficiency of Value Replacement, while still allowing the technique to yield highly effective error location results. With these improvements, it was seen that a large majority of cases required only a matter of minutes to locate each error. In a debugging context in which automated assistance can be very valuable to a developer, this timing requirement is reasonable.

The Execution Suppression technique described in this dissertation performs targeted state alteration and therefore requires significantly less computation time than Value Replacement. In most cases, just several program executions are required to isolate the first point of memory corruption in an execution that fails due to a memory error. The part of Execution Suppression that can potentially require the most computation time is the variable re-ordering step. However, the implementation heuristics described in this dissertation for significantly limiting the number of program executions required for variable re-ordering can help in improving efficiency while retaining effectiveness of the technique.

(Application to Fixing) Can dynamic state alteration techniques be applied to the problem of fixing errors?

Chapter 8 described a technique called BugFix that was inspired by the *interesting value mapping pairs* (IVMPs) computed when performing the Value Replacement state alteration technique. These IVMPs show how the values involved at particular statement instances in failing executions should be changed in order to correct the outputs of these executions. Thus, IVMPs provide important insights about what might be going wrong at a statement. However, Value Replacement by itself does not make use of this valuable information; the error location technique merely checks to see which statements are asso-

ciated with at least one IVMP. As a result, the work on BugFix is designed to make use of the insight provided by IVMPs, in order to automatically provide suggestions for how to modify a suspicious statement to fix an error. BugFix, however, takes much more information into account than just IVMPs. The technique also considers the static structure of a given statement, as well as the dynamic values used at that statement in passing and failing runs. The technique is based upon a machine-learning algorithm that can reason about prior debugging situations to provide the most relevant bug-fixing suggestions for a current debugging situation. A case study was presented that illustrates the use and potential benefit of BugFix.

10.2 Future Directions

Future directions for the work described in this dissertation involve enhancements to the presented techniques for error location, as well as applications to other areas related to handling software errors, such as fixing errors and tolerating the effects of errors.

Enhancements to Value Replacement

Although this dissertation proposed several techniques for significantly improving the efficiency of Value Replacement, there are still situations in which the technique may not scale well, such as when debugging long-running server programs that may involve very long execution traces. In cases such as this, other techniques can be used to first isolate parts of the execution that are likely to be associated with an error, and then Value Replacement can be subsequently used to rank those statements to better isolate the ones that are truly faulty. For instance, dynamic slicing techniques can first be used to identify

a subset of statement instances to later consider for value replacements. The technique of *Execution Reduction* [131] can also be used to isolate the portions of execution in long-running programs that are relevant to an error; value replacements can then be performed in only the identified portions of an execution that are relevant to the error.

Value Replacement may also have limited effectiveness in cases where it is difficult to cause the output of a failing execution to become correct, such as executions that output decimal values to fine precision. In cases such as this, it may be worthwhile to try *relaxing* the definition of an IVMP such that IVMPs are found not only when output changes to become fully correct, but also if only a subset of the output values change to become correct. Future work should consider this issue.

Finally, Value Replacement assumes that each error can be located in a single source code statement. Although this is true for many errors, in general, a software error may span multiple statements. Future work should consider multi-statement errors and how they influence the behavior of Value Replacement. Perhaps the generalized version of Value Replacement that is designed to handle multiple simultaneous errors can help in this regard.

Enhancements to Execution Suppression

The fundamental assumption made by the Execution Suppression technique is that whenever an execution involves memory corruption, that execution will produce some kind of failure (such as a crash) that will reveal a subset of this memory corruption. In general, this assumption does not always hold, and this can cause the technique to terminate prematurely and not identify the first point of memory corruption in an execution. The

technique of variable re-ordering was proposed to help address this issue, because in some cases, variable re-ordering can expose a crash due to memory corruption in cases where a crash may not otherwise occur. However, variable re-ordering can potentially be time-consuming, even with the heuristics for improving efficiency proposed in this dissertation. Future work should consider other ways of improving the efficiency of variable re-ordering, and to evaluate the resulting effectiveness of the re-ordering technique. Moreover, other techniques besides variable re-ordering should be explored in conjunction with Execution Suppression for revealing subsets of memory corruption. For instance, *AccMon* [155] could be used to identify memory corruption in cases where an outlier instruction accesses a particular memory location.

Future work on Execution Suppression should also enhance the technique to work effectively on other kinds of memory errors that may not involve memory corruption, such as memory leaks. By combining these enhancements with the proposed technique for handling multithreading memory errors including data races, the Execution Suppression technique can become very general and powerful, working effectively for all major types of memory errors.

Applications to Fixing Errors

The work on BugFix discussed in this dissertation is an application of using dynamic state alteration techniques to provide automated assistance for modifying software to fix errors. Future work should include a detailed empirical evaluation of BugFix, including an evaluation of the relative effectiveness of considering each of the different types of situation descriptors. Additional types of situation descriptors should also be considered,

including descriptors associated with the failure manifested by the program, as well as descriptors associated with the context of the suspicious statement being considered, e.g., the block of code containing the suspicious statement.

Since there is so little prior work that directly focuses on fixing software errors, there is great potential for further research in this area that can have significant impact on improving the efficiency of the debugging process. Moreover, using dynamic state alteration techniques can benefit the task of fixing errors, because state alterations of failing executions can provide valuable insights into how those executions should be changed in order to make them passing. While BugFix provides automated assistance by providing suggestions for how to modify suspicious program statements, future work should consider more aggressive techniqes that automatically attempt different program modifications and then execute all available test cases to see if any modification makes all test cases pass. Such techniques can provide more direct assistance for developers by only reporting those program modifications that are guaranteed to allow all available test cases to pass.

Applications to Tolerating the Effects of Errors

Execution Suppression can be directly applied to the problem of avoiding and recovering from the effects of errors, because the basic nature of suppression involves avoiding the effects of those statement instances during execution that are directly involved in a program failure. Recent work [99] shows how suppression can be used to recover from failures in server programs. Future work should consider other ways in which suppression and other dynamic state alteration techniques can be used to avoid or recover from the negative effects of software errors.

Bibliography

[1] R. Abraham and M. Erwig. Goal-directed debugging of spreadsheets. *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 37–44, September 2005.

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. A dynamic modeling approach to software multiple-fault localization. *Proceedings of the 19th International Workshop on Principles of Diagnosis*, pages 7–14, September 2008.

[3] H. Agrawal and J. R. Horgan. Dynamic program slicing. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–256, June 1990.

[4] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London. Incremental regression testing. *IEEE International Conference on Software Maintenance*, pages 348–357, September 1993.

[5] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. *Proceedings of the 6th IEEE International Symposium on Software Reliability Engineering*, pages 143–151, October 1995.

[6] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 22(2):207–216, 1993.

[7] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, 1994.

[8] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. *2008 IEEE Symposium on Security and Privacy*, pages 263–277, May 2008.

[9] P. Ammann and J. Knight. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers*, 37(4):418–425, April 1988.

[10] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, January 2002.

[11] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, pages 103–122, May 2001.

[12] S. P. Bear and T. Rush. Rigorous software engineering: A method for preventing software defects. *Hewlett-Packard Journal*, December 1991. http://findarticles.com/p/articles/mi_m0HPJ/is_n5_v42/ai_11648680/.

[13] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 158–168, June 2006.

[14] M. Bond. Diagnosing and tolerating bugs in deployed systems. *Ph.D. Thesis, The University of Texas at Austin*, 2008.

[15] M. Bond and K. McKinley. Tolerating memory leaks. *23rd Annual International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 109–126, October 2008.

[16] M. Bond and K. McKinley. Leak pruning. *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 277–288, March 2009.

[17] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, June 2000.

[18] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – a technique for cheap recovery. *Proceedings of the 6th Symposium on Opearting Systems Design and Implementation*, pages 31–44, December 2004.

[19] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 378–387, June/July 2005.

[20] T. Y. Chen and Y. Y. Cheung. Dynamic program dicing. *Proceedings of the IEEE International Conference on Software Maintenance*, pages 378–385, September 1993.

[21] M. Christiaens and K. D. Bosschere. TRaDe, a topological approach to on-the-fly race detection in java programs. *Proceedings of the Java Virtual Machine Research and Technology Symposium*, pages 105–116, 2001.

[22] H. Cleve and A. Zeller. Locating causes of program failures. *27th International Conference on Software Engineering*, pages 342–351, May 2005.

[23] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. *Proceedings of the 27th ACM/IEEE International Conference on Software Engineering*, pages 422–431, May 2005.

[24] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 482–493, June 2007.

[25] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96, May 1991.

[26] M. Dowson. The Ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84, March 1997.

[27] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. *Proceedings of the 16th Symposium on Operating Systems Principles*, pages 57–72, October 2001.

[28] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[29] D. Evans. Static detection of dynamic memory errors. *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 44–53, May 1996.

[30] L. Fei and S. P. Midkiff. Artemis: Practical runtime monitoring of applications for execution anomalies. *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 84–95, June 2006.

[31] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *Science of Computer Programming*, 71(2):89–109, April 2008.

[32] C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theoretical Computer Science*, 338(1-3):153–183, June 2005.

[33] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245, June 2002.

[34] J. S. Foster, R. Johnson, J. Kodumal, and A. Aiken. Flow-insensitive type qualifiers. *ACM Transactions on Programming Languages and Systems*, 28(6):1035–1087, November 2006.

[35] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 1–12, June 2002.

[36] J. Gray. Why do computers stop and what can be done about it? *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, January 1986.

[37] A. Groce. Error explanation and fault localization with distance metrics. *Ph.D. Thesis, Carnegie Mellon University*, 2005.

[38] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer*, 8(3):229–247, June 2006.

[39] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. *IEEE/ACM International Conference on Automated Software Engineering*, pages 263–272, November 2005.

[40] T. Gyimothy, A. Beszedes, and I. Forgacs. An efficient relevant slicing method for debugging. *ACM/SIGSOFT Foundations of Software Engineering*, pages 303–321, September 1999.

[41] C. Hammer, M. Grimme, and J. Krinke. Dynamic path conditions in dependence graphs. *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 58–67, January 2006.

[42] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. *Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, May 2002.

[43] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, September 2000.

[44] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 83–90, June 1998.

[45] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. *Proceedings of the USENIX Winter Technical Conference*, pages 125–136, 1992.

[46] H. He. Automated debugging using path-based weakest preconditions. *Master's Thesis, The University of Arizona*, 2004.

[47] H. He and N. Gupta. Automated debugging using path-based weakest preconditions. *Fundamental Approaches to Software Engineering*, pages 267–280, March 2004.

[48] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. *Proceedings of the 10th International SPIN Workshop on Model Checking of Software*, pages 235–239, May 2003.

[49] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, December 2004.

[50] `http://bugs.gentoo.org/show_bug.cgi?id=70090`. Prozilla error description.

[51] `http://bugs.mysql.com/bug.php?id=110`. MySQL error description.

[52] `http://bugs.mysql.com/bug.php?id=169`. MySQL error description.

[53] `http://bugs.mysql.com/bug.php?id=791`. MySQL error description.

[54] `http://marsprogram.jpl.nasa.gov/msp98/news/mco990930.html`. Mars climate orbiter team finds likely cause of loss.

[55] http://valgrind.org. Valgrind homepage.

[56] http://www.borgelt.net/apriori.html. Apriori implementation.

[57] http://www.computerworld.com/s/article/print/83735/Why_Aren_t_We_Doing_More_to_Prevent_Errors_. Why aren't we doing more to prevent errors?

[58] http://www.cse.unl.edu/~galileo/sir. Software-artifact infrastructure repository.

[59] http://www.dwheeler.com/sloccount. SLOCCount homepage.

[60] http://www.nist.gov/public_affairs/releases/n02-10.htm. Software errors cost U.S. economy $59.5 billion annually.

[61] http://www.nytimes.com/2000/07/28/us/john-tukey-85-statistician-coined-the-word-software.html. John Tukey, 85, statistician; coined the word 'software'.

[62] http://www.securityfocus.com/bid/12635. Prozilla error description.

[63] http://www.securityfocus.com/bid/13059. Axel error description.

[64] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow and controlflow-based test adequacy criteria. *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200, May 1994.

[65] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. *SIGSOFT Software Engineering Notes*, 25(5):14–25, September 2000.

[66] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta. Bugfix: A learning-based tool to assist developers in fixing bugs. *17th IEEE International Conference on Program Comprehension*, pages 70–79, May 2009.

[67] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. *International Symposium on Software Testing and Analysis*, pages 167–178, July 2008.

[68] D. Jeffrey, N. Gupta, and R. Gupta. Identifying the root causes of memory bugs using corrupted memory location suppression. *IEEE International Conference on Software Maintenance*, pages 356–365, September 2008.

[69] D. Jeffrey, N. Gupta, and R. Gupta. Effective and efficient localization of multiple faults using value replacement. *IEEE International Conference on Software Maintenance*, September 2009.

[70] D. Jeffrey and R. Gupta. Isolating multithreading bugs using execution suppression. *In submission (journal).*

[71] D. Jeffrey, V. Nagarajan, N. Gupta, and R. Gupta. Execution suppression: An automated iterative technique for locating memory errors. *In submission (journal).*

[72] L. Jiang and Z. Su. Context-aware statistical debugging: From bug predictors to faulty control flow paths. *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 184–193, November 2007.

[73] J. A. Jones, J. F. Bowring, and M. J. Harrold. Debugging in parallel. *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 16–26, July 2007.

[74] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282, November 2005.

[75] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. *Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, May 2002.

[76] M. Kamkar. Interprocedural dynamic slicing with applications to debugging and testing. *Ph.D. Thesis, Linkoping University*, 1993.

[77] A. Ko and B. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. *Proceedings of the 30th International Conference on Software Engineering*, pages 301–310, May 2008.

[78] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[79] B. Korel and J. Laski. Algorithmic software fault localization. *Annual Hawaii International Conference on System Sciences*, pages 246–252, January 1991.

[80] B. Korel and J. Rilling. Application of dynamic slicing in program debugging. *Proceedings of the International Symposium on Automated Analysis-Driven Debugging*, pages 43–58, May 1997.

[81] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing data races on-the-fly. *Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*, pages 54–64, July 2007.

[82] J. Krinke. Visualization of program dependence and slices. *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 168–177, September 2004.

[83] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, March 2006.

[84] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. *Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 306–315, September 2005.

[85] B. Liblit. Cooperative bug isolation. *Ph.D. Thesis, The University of California, Berkeley*, 2004.

[86] B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug isolation via remote program sampling. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 141–154, June 2003.

[87] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–26, June 2005.

[88] C. Liu, X. Yan, L. Fei, J. Han, and S. Midkiff. SOBER: Statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes*, 30(5):286–295, September 2005.

[89] B. Livshits and T. Zimmermann. DynaMine: Finding common error patterns by mining software revision histories. *Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 296–305, September 2005.

[90] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. BugBench: Benchmarks for evaluating bug detection tools. *Workshop on the Evaluation of Software Defect Detection Tools Co-located with PLDI*, June 2005.

[91] S. Lu, P. Zhou, W. Liu, Y. Zhou, and J. Torrellas. PathExpander: Architectural support for increasing the path coverage of dynamic bug detection. *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 38–52, December 2006.

[92] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: Trading address space for reliability and security. *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–124, March 2008.

[93] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: Explaining program failures via postmortem static analysis. *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 63–72, November 2004.

[94] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 24–33, November 1991.

[95] G. Misherghi and Z. Su. HDD: Hierarchical delta debugging. *Proceedings of the 28th International Conference on Software Engineering*, pages 142–151, May 2006.

[96] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. The soft error problem: An architectural perspective. *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 243–247, February 2005.

[97] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 446–455, June 2007.

[98] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. *8th USENIX Symposium on Operating Systems Design and Implementation*, pages 267–280, December 2008.

[99] V. Nagarajan, D. Jeffrey, and R. Gupta. Self-recovery in server programs. *Proceedings of the 2009 International Symposium on Memory Management*, pages 49–58, June 2009.

[100] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 284–295, June 2005.

[101] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. *International Conference on Programming Language Design and Implementation*, pages 22–31, June 2007.

[102] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. *Symposium on Principles of Programming Languages*, pages 128–139, January 2002.

[103] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 89–100, June 2007.

[104] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–11, June 2007.

[105] R. O'Callahan and J. D. Choi. Hybrid dynamic data race detection. *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 167–178, June 2003.

[106] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. *Object-Oriented Programming, 19th European Conference*, pages 504–527, July 2005.

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

[108] K. Pattabiraman, V. Grover, and B. Zorn. Samurai: Protecting critical data in unsafe languages. *EuroSys 2008*, pages 219–232, April 2008.

[109] D. A. Patterson. Recovery oriented computing: A new research agenda for a new century. *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 247, February 2002.

[110] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. *Computer Science Technical Report UCB//CSD-02-1175, University of California, Berkeley*, 2002.

[111] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, October 1998.

[112] M. Prvulovic and J. Torrelas. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. *30th Annual International Symposium on Computer Architecture*, pages 110–121, June 2003.

[113] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, April 2005.

[114] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 291–302, February 2005.

[115] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies – a safe method to survive software failures. *ACM Transactions on Computer Systems*, 25(3):Article 7 (1–33), August 2007.

[116] B. Randell, P. A. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys*, 10(2):123–165, June 1978.

[117] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. *Proceedings of the 27th International Symposium on Computer Architecture*, pages 25–36, June 2000.

[118] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. *International Symposium on Code Generation and Optimization*, pages 243–254, March 2005.

[119] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 30–39, October 2003.

[120] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. *Proceedings of the 6th Symposium on Opearting Systems Design and Implementation*, pages 303–316, December 2004.

[121] M. Ronsse and K. D. Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, November 1999.

[122] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, February 2004.

[123] Y. Saito. Jockey: A user-space library for record-replay debugging. *Proceedings of the 6th International Symposium on Automated and Analysis-Driven Debugging*, pages 69–76, September 2005.

[124] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[125] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. HeapMon: A helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *IBM Journal of Research and Development*, 50(2/3):261–275, March 2006.

[126] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. *USENIX Annual Technical Conference*, pages 149–161, April 2005.

[127] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. *USENIX Annual Technical Conference*, pages 29–44, June/July 2004.

[128] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems*, 24(4):333–360, November 2006.

[129] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 207–222, October 2003.

[130] S. Tallam, C. Tian, and R. Gupta. Dynamic slicing of multithreaded programs for race detection. *International Conference on Software Maintenance*, pages 97–106, September 2008.

[131] S. Tallam, C. Tian, R. Gupta, and X. Zhang. Enabling tracing of long-running multithreaded programs via dynamic execution reduction. *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 207–218, July 2007.

[132] S. Tallam, C. Tian, R. Gupta, and X. Zhang. Avoiding program failures through safe execution perturbations. *32nd Annual IEEE International Computer Software and Applications Conference*, pages 152–159, July 2008.

[133] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[134] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: A programmable accelerator for dynamic taint propagation. *Proceedings of the 14th IEEE International Symposium on High-Performance Computer Architecture*, pages 173–184, February 2008.

[135] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. *Proceedings of the 29th International Symposium on Computer Architecture*, pages 87–98, May 2002.

[136] C. Wang, H. Kim, Y. Wu, and V. Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. *Proceedings of the 2007 International Symposium on Code Generation and Optimization*, pages 244–258, March 2007.

[137] T. Wang and A. Roychoudhury. Using compressed bytecode traces for slicing java programs. *Proceedings of the 26th International Conference on Software Engineering*, pages 512–521, May 2004.

[138] T. Wang and A. Roychoudhury. Automated path generation for software fault localization. *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 347–351, November 2005.

[139] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[140] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 351–363, January 2005.

[141] Y. Xie and D. Engler. Using redundancies to find errors. *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 51–60, November 2002.

[142] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *ACM SIGSOFT Software Enginerring Notes*, 30(2):1–36, March 2005.

[143] J. Yang, C. Sar, and D. R. Engler. EXPLODE: A lightweight, general system for finding serious storage system errors. *7th Symposium on Operating Systems Design and Implementation*, pages 131–146, November 2006.

[144] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. *ACM SIGOPS Operating Systems Review*, 39(5):221–234, December 2005.

[145] A. Zeller. Isolating cause-effect chains from computer programs. *10th International Symposium on the Foundations of Software Engineering*, pages 1–10, November 2002.

[146] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.

[147] X. Zhang. Fault location via precise dynamic slicing. *Ph.D. Thesis, The University of Arizona*, 2006.

[148] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. *Proceedings of the 28th International Conference on Software Engineering*, pages 272–281, May 2006.

[149] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 169–180, June 2006.

[150] X. Zhang and R. Gupta. Cost effective dynamic program slicing. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 94–106, June 2004.

[151] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. *IEEE/ACM International Conference on Software Engineering*, pages 319–329, May 2003.

[152] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. *Proceedings of the 6th International Symposium on Automated Analysis-Driven Debugging*, pages 33–42, September 2005.

[153] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 81–91, November 2006.

[154] M. Zhivich and R. K. Cunningham. The real cost of software errors. *IEEE Security and Privacy*, 7(2):87–90, March/April 2009.

[155] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. P. Midkiff, and J. Torrellas. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. *37th Annual International Symposium on Microarchitecture*, pages 269–280, December 2004.

[156] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-assisted lockset-based race detection. *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture*, pages 121–132, February 2007.