Testing in Resource-Constrained Environments

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

---

in partial fulfillment

of the requirements for the degree

Doctor of Philosophy

by

Kristen R. Walcott-Justice

May

2012

APPROVAL SHEET

The dissertation

is submitted in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Kristen R. Walcott - Justice

AUTHOR

The dissertation has been read and approved by the examining committee:

Advisor

Accepted for the School of Engineering and Applied Science:

James H. Ay

Dean, School of Engineering and Applied Science

May

2012

Abstract

In recent software development environments, resources including time, memory, and power constrain the execution of software test cases. Often the cost of running all test cases is prohibitively expensive, necessitating techniques to select test cases of the highest quality for test execution. While many test case selection and prioritization techniques exist to select test cases that are more likely to find faults, none take the resource constraints under which the tests will operate directly into consideration. Another cost in test case execution comes from structural testing. Structural testing is an important technique in which during execution test cases are also evaluated for structural coverage. This is typically performed through the use of code instrumentation. However, instrumentation incurs high overhead in terms of time and code growth, making it difficult to use in resource-constrained environments such as mobile device testing.

This thesis develops and evaluates techniques for efficient and effective execution of test cases in resource-constrained environments. The first set of techniques selects high quality test cases from test suites while taking resource constraints into account. The resulting test selection is guaranteed to execute within the specified resource constraints, and the resulting test cases have the highest possible potential for overall fault detection during their constrained execution. The selected tests are additionally ordered in a way that is likely to find faults earlier in test execution rather than later. These techniques represent the first to guarantee that test selections will execute within a given constraint, and they offer a spectrum of options for selecting tests to execute in resource-constrained environments.

The second set of techniques evaluates the quality of test cases during execution with-

out the need of expensive and intrusive instrumentation. Instead, our quality evaluation techniques exploit recent hardware advances to monitor program execution. By leveraging these hardware advances, up to 90% of branch coverage and 79% of statement coverage can be determined with less time overhead compared to instrumentation. Because the techniques require little or no code modification or dynamic code insertion and can run on commodity machines, tablets, or mobile devices, the techniques also enable test case execution and evaluation on resource-constrained devices. No specialized hardware or components are required. The techniques are combined into a runtime and static testing system called THeME: Testing by Hardware Monitoring Events. THeME is a portable, extensible system, making it applicable for use in a wide range of resource-constrained environments.

Acknowledgements

I would like to take this opportunity to thank the people who have helped me through this journey. First, the role played by my advisor, Mary Lou Soffa, cannot be expressed in just a few words. She has been a wonderful mentor, teacher, and friend. I cannot thank her enough for her support, advice, encouragement, and patience.

I would also like to thank Greg Kapfhammer for all of his help and advice over the years. Greg has been a key role model for me in terms of both research and teaching, and he is an incredible teacher and mentor. I greatly appreciate all of the effort he has put into supporting me. I truly look forward to continuing our collaboration in the future.

Sudhanva Gurumurthi also has provided me with a lot of help and encouragement. I worked with Sudhanva for about a year and a half while completing my masters, and I greatly enjoyed working with him. Thank you for your time and advice.

Thanks also are certainly due to Chris Lauderdale and Dan Upton. Chris and Dan both have been wonderful supporters and friends for as long as I've known them, and I cannot thank them enough. Also, to my fellow graduate students and colleagues, particularly Jason Mars, Lingjia Tang, Tanima Dey, Wei Wang, Jing Yang, Sudeep Ghosh, and Karolina Sarnowska-Upton, thanks for all of your help and for putting up with me.

Last, thanks go out to my wonderful husband, Beau Justice, my in-laws, Tom and Ginny Justice, and my parents, Richard and Kathleen Walcott. They have stood by me every step of the way, providing support and encouragement- as well as multiple important lessons on comma usage. I could not have done this without them, and I very much appreciate everything they've done.

Contents

Abstract i

Acknowledgements iii

List of Figures viii

List of Tables xi

1 Introduction 1

1.1 Test Case Selection . . . 2

1.2 Executing Test Cases . . . 4

1.3 Constraints During Testing . . . 4

1.3.1 Constraints When Selecting Test Cases . . . 5

1.3.2 Constraints When Executing Test Cases . . . 7

1.4 Challenges and Goals of Testing in Resource-Constrained Environments . . . 8

1.5 Research Overview . . . 11

1.5.1 Description of the Research Process . . . 12

1.5.2 Contributions of the Dissertation . . . 15

2 Background and Related Work 17

2.1 Test Suite Design and Analysis . . . 17

2.1.1 Evaluating Test Suite Quality . . . 18

2.1.2 Measuring Test Suite Effectiveness . . . 20

2.2 Related Work . . . 22
2.2.1 Test Selection and Prioritization . . . 22
2.2.2 Executing Test Cases Efficiently . . . 25
2.2.3 Hardware Performance Monitoring and Sampling . . . 26

3 Knapsack Solvers for Time-Aware Selection 28
3.1 Time-Aware Selection . . . 30
3.2 Knapsack Solvers as Selectors . . . 30
3.3 Experiment Goals and Design . . . 34
3.3.1 Case Studies . . . 35
3.3.2 Evaluation Metrics . . . 36
3.4 Experiments and Results . . . 40
3.4.1 Selection Effectiveness. . . . 40
3.4.2 Selection Efficiency. . . . 42
3.5 Conclusions . . . 44

4 A Genetic Algorithm for Time-Aware Selection 45
4.1 Genetic Algorithms and the Test Selection Challenge . . . 46
4.1.1 Designing a Genetic Algorithm . . . 46
4.1.2 Genetic Algorithm Challenges . . . 48
4.1.3 The Test Selection Challenge . . . 49
4.2 Time-Constrained Selection . . . 49
4.2.1 Overview . . . 50
4.2.2 A Genetic Algorithm for Time-Aware Test Selection . . . 50
4.2.3 Test Selection in Action . . . 57
4.3 Empirical Evaluation . . . 59
4.3.1 Experimental Design . . . 60
4.3.2 Experiments and Results . . . 63
4.4 Conclusions . . . 75

5 Executing Test Cases for Branch Monitoring 77

5.1 Challenges of Exploiting Hardware Mechanisms . . . 80

5.2 Hardware Monitoring for Branch Testing . . . 82

5.2.1 Last Branch Record (LBR) . . . 82

5.2.2 Branch Trace Store (BTS) . . . 84

5.3 Sampling Hardware Mechanisms for Branch Testing . . . 84

5.3.1 Event-Based Sampling . . . 85

5.3.2 Addressing the Challenges of Sampling . . . 85

5.3.3 Improving Effectiveness with MultiCores . . . 88

5.4 Empirical Evaluation . . . 88

5.4.1 Experiment Design . . . 89

5.4.2 Experiments and Results . . . 92

5.5 Conclusion . . . 100

6 THeME: Testing by HardwareMonitoring Events 102

6.1 Improvement Challenges . . . 103

6.2 Accessing and Sampling Branch Vectors . . . 104

6.2.1 Implementation Details . . . 104

6.2.2 User-level Branch Vector Access . . . 106

6.2.3 Access via Polling . . . 106

6.2.4 Interrupt Driven Access . . . 107

6.3 Improving Branch Coverage at High Sample Rates . . . 110

6.4 Testing over Multiple Cores . . . 111

6.5 Discussion . . . 113

6.5.1 LBR Monitoring Benefits . . . 113

6.5.2 Improving Efficiency by Advancing Hardware . . . 114

6.5.3 Improving Effectiveness Through Instrumentation . . . 117

6.6 Conclusion . . . 118

7 Executing Test Cases for Statement Monitoring 120

7.1 Challenges of Statement Monitoring 122

7.2 Hardware Monitoring for Statement Testing 123

7.2.1 Choosing a Mechanism 124

7.2.2 Statement Coverage Comparisons 125

7.3 Discussion 127

7.4 THeME for Tablets and Smartphones 128

8 Merits and Future Work 131

8.1 Contributions and Merits 132

8.2 Future Work 133

8.2.1 Selecting Test Cases Based on Estimations 134

8.2.2 Combining Hardware Sampling and Limited Instrumentation 134

8.2.3 Execution for Evaluation of Other Test Metrics 134

8.2.4 Hardware Mechanisms and VM Environments 136

8.2.5 Fault Localization 136

Bibliography 137

List of Figures

1.1 Example Test Suite Selections Using a Naïve Selection and a Constraint-Aware Selection. 6

1.2 Research Process- The Five Components of the Thesis Research 12

3.1 Generalized Tabular Example. 32

3.2 Example Test Cases. 33

3.3 Scaling Heuristic Example. 33

3.4 Overview of the Selection Infrastructure. 34

3.5 Case Study Applications. 35

3.6 $\langle T_2, T_1 \rangle$ covers more requirements early in execution than $\langle T_1, T_2 \rangle$. 37

3.7 Coverage Preservation of Test Suite Selection. 41

3.8 Overall Coverage and Order-Aware Coverage of Test Suite Selection. 41

3.9 Memory Overhead of Test Suite Selection. 43

3.10 Time Overhead of Test Suite Selection. 43

4.1 Genetic Algorithm Procedure. 47

4.2 The GA Selection Algorithm. 51

4.3 Crossover with Random Crossover Point. 55

4.4 Mutation of a Test Tuple. 56

4.5 Overview of Selection Infrastructure. 60

4.6 GA Coverage Preservation, Coverage/Fitness, and APFD Results. 65

4.7 Genetic Algorithm Time Results. 67

4.8 The Complete Selection Generator. 70
4.9 GA vs. Random Ordering APFD Values. 72
4.10 Coverage Preservation of Test Suite Selection. 74
4.11 Genetic Algorithm Comparison to Non-Overlap-Aware 0/1 Knapsack Solvers. 74

5.1 64-bit Layout of the LBR MSR [47]. 82
5.2 The Debug Store Area [47]. 83
5.3 The LBR is incapable of detecting the fall-through branch edge from 1 to 2. 86
5.4 Overview of infrastructure to adapt LBR monitoring to branch testing. . . 89
5.5 Time overhead for event-based sampling on a single core relative to full instrumentation. 95
5.6 Percent of actual coverage from event-based sampling on single and multiple cores. 96
5.7 Time overhead relative to full instrumentation of a simulation of using a filtering mechanism. 98
5.8 Percent of actual coverage obtained during a simulation of using a filtering mechanism. 98

6.1 Infrastructure to adapt LBR monitoring to branch testing. 104
6.2 Time overhead for LBR sampling accessed using polling relative to full instrumentation. 107
6.3 Time overhead for LBR sampling accessed using an interrupt-driven approach relative to full instrumentation on `test` inputs. 108
6.4 Time overhead for LBR sampling accessed using an interrupt-driven approach relative to full instrumentation on `ref` inputs. 109
6.5 Coverage observed using LBR sampling via the interrupt-driven approach on `ref` inputs compared to instrumentation. 110
6.6 Dominator analyses based on an observed branch. 111

6.7 Time overhead for LBR sampling over multiple cores compared to using instrumentation on multiple cores. 113
6.8 Moving private performance monitoring units to a global space to enable Satellite Monitoring. 115
6.9 Percent of coverage observed when selectively instrumenting branches compared to instrumentation. 117

7.1 Infrastructure to adapt CPU_CYCLE monitoring to statement testing. . . . 123
7.2 Time overhead for Instructions Retired compared to native on `test` inputs. 124
7.3 Time overhead for CPU Cycles compared to native on `test` inputs. 124
7.4 Statement Coverage using Instructions Retired on `test` inputs. 125
7.5 Statement Coverage using CPU Cycles on `test` inputs. 125
7.6 Time overhead relative to full instrumentation on *ref* inputs 126
7.7 Statement coverage using CPU Cycles compared to Instrumentation on *ref* 126
7.8 The Cross Trigger Interface [12] 129

List of Tables

2.1 Faults detected by test suite $T = \langle T_1, \ldots, T_7 \rangle$. 21

4.1 GA Problem Formulation and Configuration. 50
4.2 Faults Detected by $T = \langle T_1, \ldots, T_7 \rangle$. 61
4.3 Parameters used in GA Configurations. 64
4.4 `Gradebook` and `JDepend` APFD Values. 65
4.5 Initial, Reverse, Fault-Aware, and Genetic Algorithm Selection APFD Values. 73

5.1 Number of branch edges and actual branch coverage of original program calculated using a software based instrumentation tool. 92
5.2 Observed BTS overhead on a subset of the SPEC2006 benchmarks. 93
5.3 Time overheads & code size of native, fall-through enabled, and software-instrumented benchmarks using *test* inputs. 94

6.1 SPEC 2006 benchmark time overhead information. 105

7.1 Code Growth of Instrumentation vs CPU Cycles 126

Chapter 1

Introduction

Software testing is a critical and expensive component of the software development lifecycle, accounting for 50-60% of the total cost of software development [81]. Due to the large number of test cases that are created during the software life cycle, the cost of executing and evaluating all tests is often prohibitive. Executing an entire test suite during the testing or regression testing phases of development may require days, weeks, or even months of time, often with expensive equipment and engineering costs associated. As shown in an example in [28], an industrial collaborator reported that for one of its products of approximately 20,000 lines of code, the entire test suite required seven weeks to run. Due to the substantial amount of overhead incurred, developers continually seek methods to reduce the cost of testing.

Much of the cost of software testing comes from executing the entire existing test suite. One way to reduce the cost of testing is to reduce the set of test cases that will be run. Reduction techniques include test suite minimization and test case selection. When reducing the number of test cases that will be run, care must be taken not to jeopardize the quality of the test suite. Additionally, to ensure that defects can be determined as early as possible in the testing process, prioritization techniques can be included.

Another source of cost results from evaluating the test suite during its execution. A common indicator for evaluating the quality of test cases is structural code coverage. This concept refers to the portion of a software source code that is actually executed during a

particular run. Traditionally, tests are evaluated and monitored through the insertion and execution of instrumentation that is inserted within the source code or program binary. To instrument code, the program is analyzed to determine code points of interest. Each point is marked by a probe, which is usually a jump or call to payload code that analyzes the monitored information. The time overhead and code growth from instrumentation can be high, even when monitoring simple structures. For example, the time overhead of using instrumentation for monitoring branches has been reported to be, on average, between 10% to 30%, with code growth ranging from 60% to 90% [76, 95, 106].

Given the recent trends of developing applications for mobile devices, of creating larger, more complex systems, and of testing software more frequently in short development cycles, there are greater challenges when testing software due to the constraints that each of these tasks entail. These tasks necessitate that constraints such as time, power, and memory are taken into account. When testing software on mobile devices, power and memory constraints must be met while maintaining a high level of fault finding capability. Testing large, complex systems and testing in short development cycles bring time constraints to the forefront, but the ability of the test suite to detect faults should also be optimized. In this dissertation, we develop techniques for efficient and effective software testing within resource-constrained environments.

1.1 Test Case Selection

Due to the high costs of executing all tests within a test suite, techniques have been developed to reduce the number of test cases that must be executed while maintaining a level of quality exhibited by the original test suite. Test suite minimization techniques attempt to reduce the overall size of the test suite by removing obsolete and redundant test cases without reducing fault finding capability [41, 88, 119]. A change in a program causes a test case to become obsolete if the reason for the test cases inclusion in the test suite is removed. A test case is redundant if other test cases in the test suite test the same functionality

within the program. Reduction in the size of the test suite decreases both the overhead of maintaining the test suite and the number of test cases that must be rerun as changes are made to the software.

Test suite selection techniques further attempt to reduce the cost of testing by selecting and running only a subset of the test cases in an existing test suite. Selection techniques are generally based on information about the program, modified versions of the program, and the existing test suite [16, 24, 63, 87].

While minimization and selection approaches may lessen the cost of performing testing, it is quite difficult to find a balance between the time required to select and run test cases and the fault detection capability of the remaining test cases. Although some empirical evidence indicates that there is little or no loss in the capability of a reduced or selected test suite to reveal faults in comparison to the original test suite [119, 120], other empirical evidence shows that that the fault detection capabilities of reduced test suites can be severely compromised [88]. Safe test selection techniques do exist (e.g., the technique created by Rothermel and Harrold [87]), but the work required to prove that the subset of test cases exposes the same number of faults as the full test suite is difficult. Also, the conditions under which safety can be achieved do not always hold [86, 87].

Test case prioritization techniques can additionally be used to execute test cases in an order that increases their effectiveness at achieving some goal. For example, test cases might be ordered so that those with the highest levels of estimated fault-finding capability are run first [90]. Usually tests are arranged to improve the rate of fault detection, a metric that reveals how quickly faults are detected during a test case's execution [28]. An improved rate of fault detection during testing can provide faster feedback on a system under test and let debuggers begin their work earlier than might otherwise be possible.

Prioritization techniques are often preferred to selection techniques because they do not eliminate any tests from the initial test suite. If test case execution is halted early, prioritization techniques ensure that test cases that offer the greatest fault detection capability will have been executed.

1.2 Executing Test Cases

After tests are generated and selected during execution, they must also be evaluated to determine the quality of the executing tests. Evaluation of quality can identify sections of source code or execution paths that are not executed by the current test suite, identifying a need for additional test cases. The quality evaluation can additionally identify test cases that are redundant and thus needlessly add to the cost of test suite execution. Multiple test cases with the same estimated fault-finding capability are unlikely to benefit the testing process. Traditionally, instrumentation is used to analyze structural test coverage and to estimate the fault-finding capabilities of tests.

In most tools, instrumentation is statically placed in the source or binary code before execution and remains in the code until execution terminates. This type of instrumentation can be expensive in both time and space and requires recompilation of the program under test. Even though coverage may only require one instantiation of a code element, the instrumentation generally stays in the code, causing unnecessary time overhead. Static instrumentation may also be placed along infeasible paths, which leads to imprecision as well as avoidable code growth [17].

Dynamic instrumentation tools, such as Pin [69], add instrumentation each time a selected program element is going to be executed. A just-in-time compiler is used to compile new code for the straight line code sequence starting at the instruction to be instrumented. For each insertion, the tool transfers control to the generated sequence [69]. While dynamic techniques are advantageous in that they do not require static code modification, the time and memory overheads are generally substantially higher than that of static instrumentation [107, 111].

1.3 Constraints During Testing

In recent software development environments, the overheads of testing become more restrictive as time, memory, and power constrain the extent to which software can be tested.

In many settings, multiple constraints must be taken into account at once. Despite the constraints at hand, a high level of quality should be maintained.

1.3.1 Constraints When Selecting Test Cases

Many development environments today have greater time, power, and monetary constraints than those traditionally experienced. As one example, frequent testing of applications and short development cycles are gaining in popularity in many development environments. Popular software systems such as `Firefox` [4], `XBMC` [7], and VLC [8] use nightly builds. A nightly build is a neutral build that takes place automatically, typically executing when no one is likely to be working on the software so that there are no changes to the source code during the build. During the process, the most recent source code that is checked into the source code version control system is built and linked, and test cases are executed. The results are inspected by the arriving programmers, who generally place a priority on ensuring that recent changes to the source code have not broken the functionality of the software [73].

Short building and testing phases are also common in extreme programming [84]. Extreme programming is intended to improve software quality and responsiveness to changing customer requirements. Extreme programming is a type of agile software development, which advocates frequent releases in short development cycles.

Within these short building and testing cycles, it is generally infeasible to execute all existing test cases due to time, power consumption, and monetary constraints. Current test suite prioritization techniques allow tests to be ordered so that test cases with the highest fault finding capability are executed first, and selection techniques reduce the number of tests that must be run while maintaining a high level of quality. However, given a set of tests where each test has an estimated fault finding capability and a measure of resource usage, more effective test cases can be selected for execution when resource constraints are considered during the selection and prioritization processes.

As an example, suppose there exists a test suite with only five test cases, as described in

	Number of Faults	Time Cost	Average Faults Found per Minute
T_1	21	25	0.84
T_2	5	6	0.833
T_3	5	7	0.714
T_4	4	5	0.8
T_5	8	10	0.8

(a)

	Time Limit: 29 minutes	
	Naïve Selection	Time-Aware Selection
	T_1	T_2
		T_4
		T_5
		T_3
Total Faults	21 faults	22 faults
Total Time	25 minutes	28 minutes

(b)

Figure 1.1: Example Test Suite Selections Using a Naïve Selection and a Constraint-Aware Selection.

Figure 1.1. Test case T_1 can find twenty-one faults in twenty-five minutes. T_2 finds five faults in six minutes, T_3 isolates five faults in seven minutes, T_4 finds four faults in five minutes, and T_5 finds eight faults in ten minutes. A non-time-constrained naïve prioritization technique would run T_1, followed by T_2, then T_4 and T_5, and finally number T_3. However, in a time-constrained situation, suppose, twenty-nine minutes, T_1 alone would be run, only finding a total of twenty-one faults. Because that test had the greatest expected rate of fault detection, a typical prioritization technique would likely run T_1 first, leaving no time for any other tests. However, the test cases could be better selected and prioritized, maximizing the number of faults found in the desired time period. In this example, the test cases would be reordered so that T_2, T_4, T_5, and T_3 will run. In that order, the test suite would discover a total of twenty-two errors in the time frame.

When testing constraints are known, a constraint-aware selection technique is likely to detect more faults than more traditional approaches. Generally, prioritization and selection techniques only account for fault-finding capability. However, in many popular testing

environments, time or power also should be considered. Due to constraints, some tests may need to be removed from the original test suite, but the selected test cases should maintain as high a quality as possible while also detecting faults as early in the testing process as possible.

1.3.2 Constraints When Executing Test Cases

Traditional techniques for executing test cases and evaluating test case quality generally lead to high time and memory overheads. In many cases, the overheads of using instrumentation to estimate test quality are restrictive using either static or dynamic code coverage techniques. For example, if a test suite requires 24 hours to execute, it would likely necessitate an extra 2.4 to 7.2 hours to evaluate the quality of the test suite based on branch coverage. When monitoring large scale programs or more complex structures, such as data-flow or paths, the overall cost of monitoring grows and can become prohibitive in time and space.

The memory and time overheads for executing test cases are particularly restrictive when testing applications on low memory devices including smartphones and tablet computers. Despite the popularity of such systems, there exists little support for executing tests and evaluating test quality on the devices themselves. Mobile applications are first tested within an emulator, and then they are later subjected to field testing for additional bug finding. Emulators provide an inexpensive way to test applications on mobile phones to which developers may not have physical access. However, developers recognize that when building mobile applications, it is important that the application is tested adequately on a real device before releasing it to users [1]. Given that memory on current smartphones and tablets generally ranges from 128MB to 1GB of RAM, traditional test execution and evaluation approaches using static or dynamic instrumentation can prohibitively impact the ability to test on the devices themselves due to their increased memory footprints.

A number of techniques have been developed in an attempt to reduce the time overhead of program execution. For example, in work by Arnold and Ryder [13], instrumentation

sampling is used to reduce the overhead of using complete sampling for profile collection. Their framework switches between instrumented and non-instrumented code by placing a sample condition on all method entries and backedges. A sample condition is checked, potentially causing the tool to execute fully instrumented code, based on a trigger mechanism. Using this combination of instrumented and non-instrumented code resulted in above 90% accurate profiles with 6% overhead. However, since their technique doubles all methods in size, the maximum space overhead is the sum of the sizes of the final optimized code for all methods.

When executing test cases in environments that are constrained by time or memory, techniques that are efficient in terms of both time and memory are necessary to estimate test case quality. However, these techniques should also be effective, providing an accurate estimation of test case coverage.

1.4 Challenges and Goals of Testing in Resource-Constrained Environments

The overall goal of this dissertation is to develop and evaluate testing techniques that take time and memory constraints into consideration while maintaining a high level of test quality. The first set of techniques considers the challenge of selecting and prioritizing test cases that will be executed in time-constrained settings. The second set of techniques examines novel approaches for executing test suites in time and memory-constrained environments.

Our set of techniques for selecting test cases for execution in time-constrained environments must satisfy three goals. First, our resulting set of test cases must be guaranteed to execute within set time constraints. Second, the resulting test cases should have the highest possible potential for overall fault detection. Third, the selected tests should be ordered in a way that is likely to find faults earlier in test execution rather than later.

Achieving all three of the goals within one technique is challenging. While many selection and prioritization algorithms exist, these focus only on fault-finding capability. By adding

a constraint into the equation, the test case selection problem becomes undecidable. An efficient solution to the test case selection problem would provide an efficient solution to the knapsack problem [36, 89]. The traditional knapsack problem states that given a set of items, each with a weight and a value, determine the count of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. In the test case selection problem, our goal is to maximize the estimated fault finding capability of the test suite while remaining within the constraint boundaries. Additionally, preference is given to selections that are likely to detect faults earlier in their execution. As the zero/one knapsack problem is NP-complete, there are a number of algorithms that approximate the optimal solution to this problem, which vary in complexity and optimality. Therefore, the tradeoffs between selection fault-finding capability and the efficiency to compute the selection must be analyzed.

An additional challenge is that, while test cases are independent of each other, they often cover the same or similar paths through the program. In the traditional knapsack problem, all items to be placed in the knapsack are assumed to have a static value. By adding more items, the value of the original items is expected not to change. The test suite selection problem, however, ideally can account for overlapping coverage between test cases. For example, if test cases T_1 and T_2 each cover the same code and require the same execution time, a selection including T_1 and T_2 should have no greater value than a selection including one T_1 or T_2.

In the set of techniques for executing test suites, we develop a code coverage analysis technique that has i) a reduced time overhead, ii) a reduced memory overhead, and iii) requires little or no code modification or dynamic code insertion. The technique should be able to run on commodity machines, tablets, or mobile devices, and thus should not require uncharacteristic hardware or components.

The main challenge of executing test cases is the task of monitoring program execution efficiently. Many techniques have been developed to statically or dynamically monitor program execution through the use of instrumentation. However, these techniques still

suffer from high time and memory overheads and often require the application under test to be recompiled. Static or dynamic insertion of instrumentation may also perturb normal execution.

Similar to test suite evaluation, many software development tasks such as path profiling, trace selection, race detection, and dynamic optimization are also riddled by the challenge of how to efficiently monitor application behavior. In these areas, there is an emerging trend to leverage hardware performance monitoring mechanisms and multicore technology to mitigate and eliminate these challenges [15, 20–22, 70, 85, 96]. For example, research by Chen et al. [21] shows that profile information can be constructed efficiently and effectively by sampling hardware events. In their work, event monitoring incurred runtime overhead of only 2% and no code growth compared to compiler-based instrumentation, which suffered 10x time overhead over native execution.

Compared to instrumentation, the use of hardware mechanisms is attractive as they can perform monitoring with very little overhead, and their use can remove the need for instrumentation. When monitoring using hardware mechanisms, a counter and mechanism need only be set up once per core during test execution, and reading the mechanism is inexpensive. For example, Dey et al. [26] report that the initial setup for a counter takes approximately 318μs, and reading a counter value takes only 3.5μs on average. In addition, using hardware performance monitors in lieu of instrumentation incurs no code growth and does not require recompilation.

The use of hardware mechanisms for monitoring is additionally appealing because nearly all commodity desktop, laptop, tablet, and mobile devices now available contain processors that support hardware monitoring on single and multiple cores. For example, the Intel Nehalem processor provides the capability to track more than 2000 different performance events, and recent Linux kernel patches provide user-level support for nearly 200 of these mechanisms [31]. Many processors also include advanced hardware monitoring components that can provide large amounts of event information.

Despite the success of exploiting hardware mechanisms in other software engineering

tasks, advances in hardware monitoring and multicore technology has not been fully exploited in software testing. Hardware counters can be configured on each processor core to increment when certain hardware events occur, providing count information, or they can be used for sampling. When a sample is taken, performance monitoring software records the system state including the current instruction information, register contents, etc. Such sampled information is extremely useful in areas such as profiling or dynamic optimizations because the samples can be used to estimate profiles or partial program behavior [21].

Software testing, however, relies on more exact execution information. For example, in branch testing, instrumentation is used to monitor *all* source code level branches with which the tester is concerned and monitor *only* those branches. While hardware mechanisms tracking a particular event will observe all events of that type during program execution, sampled data may miss certain events such as infrequently executed branches. Also, the use of hardware mechanisms implies that samples are likely to include branches that are not associated with the test program such as those in setup, teardown, or library code. Although recording hardware events is essentially free, there is a cost associated with reading the values from the hardware. Therefore, a balance must be found between the amount of information collected and the total overhead of sampling.

In our research, we address each of these challenges in order to meet our goals of developing effective test case selection techniques and test case execution techniques for execution in resource-constrained environments.

1.5 Research Overview

This section summarizes the thesis research. First, we introduce the five projects of the dissertation research and describe a summary of the solutions found in the research that address the targeted challenges specified in Section 1.4. Then we summarize our contributions and outline the remainder of the dissertation.

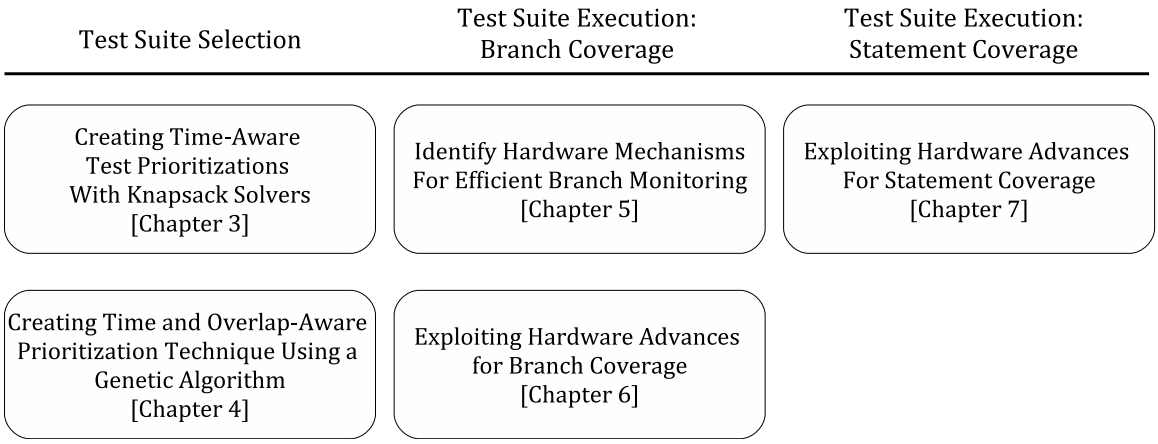


Figure 1.2: Research Process- The Five Components of the Thesis Research

1.5.1 Description of the Research Process

This research is developed in three stages, shown in Figure 1.2. In the first stage, *Test Suite Selection*, we identify and develop techniques that can be used to select test cases with regard to a given time constraint while also considering test case quality. While the likelihood of a particular test to find faults varies between different versions of the code, we assume in this work that there is no prior knowledge of test behavior or of code changes between test suite executions. Estimating fault-finding capability based on software revisions is outside the scope of this work.

We first developed seven techniques based on 0/1 knapsack approximation algorithms to select tests for execution in a time constrained environment while maintaining high test quality. Other resources such as power consumption or monetary budget constraints can be substituted for time constraints with no modification to the techniques. We found that when provided with a testing time budget, 0/1 knapsack techniques can efficiently create reduced, reordered test suites that quickly cover test requirements and always complete execution within the specified time limit. However, we also found that even the most sophisticated 0/1 knapsack solvers do not always identify the highest quality test case selections. This is due to the fact that traditional 0/1 knapsack solvers do not account for coverage overlap among

test cases. We hypothesized that a test selection technique that additionally accounts for coverage overlap will significantly improve the overall effectiveness of the final test suite but at the cost of efficiency while generating the selection.

To verify this hypothesis and to generate higher quality test suite selections, we next developed a technique that uses a genetic algorithm to select and prioritize a test suite that will run in a time constrained execution environment. In our genetic algorithm design, we rephrased the traditional 0/1 knapsack problem to account for coverage overlap between test cases. Our experimental analysis of the selections produced by the genetic algorithm demonstrate that the resulting selections lose little of the original test suite's value with regard to (i) original coverage preservation, (ii) overall coverage, and (iii) fault finding capability. We also compared the effectiveness of our resulting test case selection to those generated by current prioritization techniques that are not naturally constraint aware, revealing that the selections produced by the genetic algorithm are superior to the other test suite reorderings in the face of time constraints. We learned that while overlap-aware selectors require a larger amount of time overhead to select and order test cases, they are useful in testing contexts where correctness is the highest priority.

In the *Test Suite Execution: Branch Coverage* phase, we developed a new approach to monitoring program execution that exploits recent hardware advances for use in branch coverage analysis. We first explored how several available hardware mechanisms can be used for branch coverage analysis and identified the overheads their use requires. For test monitoring, hardware mechanisms were used to sample program execution. When a sample is taken, performance monitoring software recorded the system state including the current instruction information, register contents, etc. Some mechanisms are capable of gathering more information in fewer samples and of reporting more precise information. Thus, in this phase, we determined what hardware mechanisms are most applicable for use in test case execution when monitoring branch coverage and evaluated the benefits and disadvantages of using each mechanism within resource-constrained environments.

Regardless of the hardware mechanism selected, taking advantage of hardware mecha-

nisms leads to the challenge of determining how to gather the most complete information possible while generating acceptable overhead in terms of time and code growth. While inexpensive, there is a cost associated with accessing and reading values from hardware. Therefore, in this phase, we demonstrated the tradeoffs between the sampling rate, time overhead, and code growth that can be obtained by sampling branch-based hardware mechanisms. We learned that the code growth needed for effective monitoring is significantly smaller than that of monitoring using instrumentation, making our techniques useful particularly in memory constrained environments. Our techniques also can be used to monitor program coverage with significantly less time overhead than instrumentation.

We next made use of these mechanisms and developed a system for performing efficient branch coverage analysis during program execution. We first developed our system called THeME: Testing by Hardware Monitoring Events. THeME takes a pure hardware approach to branch testing. Analysis of our system demonstrated the efficiency achieved when calculating coverage information by sampling hardware. Additionally, we evaluated how performing branch testing using hardware sampling affects the completeness of coverage monitoring. To improve the effectiveness of our approach, we also analyzed the effects of integrating hardware monitoring information with the compiler infrastructure, which improves the completeness of coverage monitoring through the use of static analysis techniques. Finally, we explored how multiple cores can be used in conjunction with hardware monitoring to improve the time overhead of structural testing. The results show that up to 90% of the actual coverage can be determined with less time overhead and negligible code growth compared to instrumentation.

In the final phase *Test Suite Execution: Statement Coverage*, we extended THeME to execute tests while evaluating statement coverage. Statement coverage is widely used in industry as a common criterion for measuring the thoroughness of software testing [18, 25, 55, 92, 108]. However, statements are also more expensive to monitor than branches, leading to higher memory and time overheads. In our statement coverage evaluation technique, we modified THeME to make use of commonly accessible hardware mecha-

nisms that are representative of mechanisms available on commodity machines, tablets, and mobile devices alike. We evaluated THeME for statement coverage monitoring using three potential hardware mechanisms. After selecting the mechanism with the highest efficiency and effectiveness, we then analyze THeME's success at monitoring for statement coverage compared to instrumentation approaches. The results show that up to 79% of the statement coverage reported by instrumentation can be reported with lesser time overhead than instrumentation with no code growth.

Finally, we discuss the challenges of porting and executing the THeME system on tablets, smartphones, or other mobile devices on which resources are limited. Although current kernel implementations do not support hardware mechanism sampling, the statement coverage results are indicative of how the THeME system would perform on such devices. With additional kernel support, THeME can be used as a highly extensible and portable testing system that provides an efficient and effective approach to software testing.

1.5.2 Contributions of the Dissertation

The major contributions of this dissertation are the following:

1. An evaluation and discussion of the tradeoffs between time-aware test case selection quality and the efficiency of selecting using 0/1 Knapsack approximation algorithms [9, 116].

2. The development and evaluation of a time-aware test case selection infrastructure based on a genetic algorithm [115, 116].

3. An exploration and development of techniques that will evaluate the potential of exploiting hardware mechanisms in branch test coverage analysis to improve time overhead and code growth [100].

4. The development and evaluation of a runtime system, THeME, to perform efficient branch coverage analysis using hardware monitoring mechanisms and multicore technology [117].

5. The extension of THeME to perform efficient and effective statement coverage analysis.

The remainder of this dissertation is organized as follows. In Chapter 2, we discuss background information and related work. Then contribution 1 is presented in Chapter 3. Contribution 2 is presented in Chapter 4. Chapters 5 and 6 then focus on exploiting recent hardware advances for branch coverage analysis. In Chapter 5, we present contribution 3. We then apply the lessons learned in Chapter 5 to develop and evaluate the THeME system, contribution 4, which is described in Chapter 6. In Chapter 7, we present contribution 5. Finally, Chapter 8 concludes the dissertation and includes a discussion of future work.

Chapter 2

Background and Related Work

This thesis focuses on two main challenges. The first is selecting test suites for efficient and effective execution within resource-constrained environments. The second is executing the selected test cases and evaluating their quality efficiently and effectively. In this chapter, we first discuss background regarding test suites and their properties, the metrics that are commonly used to estimate test suite quality, and a metric to quantify the fault-finding capability of tests. We then present work that is related to our test case selection and test suite execution techniques.

2.1 Test Suite Design and Analysis

To assess the quality and determine the level of adequacy of an application, a test suite is executed where each test case of the test suite is designed to ensure that quality is maintained. A test suite is defined as follows [49].

Definition 1 *A test suite T is a triple $\langle \Delta_0, \langle T_1, \ldots, T_n \rangle, \langle \Delta_1, \ldots, \Delta_n \rangle \rangle$, consisting of an initial external test state, Δ_0, a test case sequence $\langle T_1, \ldots, T_n \rangle$ for some state Δ_0, and expected external test states $\langle \Delta_1, \ldots, \Delta_n \rangle$ where $\Delta_i = T_i(\Delta_{i-1})$ for $i = 1, \ldots, n$.*

Δ_i denotes the externally visible state of the application under test. Informally, Δ_i can be seen as a set of pairs made up of a variable name followed by the value for the variable

name that is used to compare the actual values computed with the expected values. Test suites are broken up into test cases that are sequences of test operations that cause the application to enter states that are visible to the specific test only.

Definition 2 *A test case* $T_i \in \langle T_1, \ldots, T_n \rangle$ *is a triple* $\langle \delta_0, \langle o_1, \ldots, o_x \rangle, \langle \delta_1, \ldots, \delta_x \rangle \rangle$*, consisting of an initial test state,* δ_0*, a test operation sequence* $\langle o_1, \ldots, o_x \rangle$ *for state* δ_0*, and expected internal test states* $\langle \delta_1, \ldots, \delta_x \rangle$ *where* $\delta_y = o_y(\delta_{y-1})$ *for* $y = 1, \ldots, x$ *[49].*

Test suites may also be independent.

Definition 3 *A test suite* T *is independent if and only if for all* $\gamma \in \{1, \ldots, n\}, \Delta_\gamma = \Delta_0$ *[49].*

Since each test case in an independent test suite returns the application under test back to the initial state, Δ_0, before it terminates, the tests can be run in any order. In this research, because test cases are being evaluated singularly and reordered, all test cases are assumed to be independent.

2.1.1 Evaluating Test Suite Quality

A fault within the program under test will manifest itself as a failure only if a test case executes the fault, causes the fault to infect the data state of the program, and propagates to the output. The fault infects the data state if it changes something within the program, for instance, a variable's value, and it propagates through to the output if the output is incorrect [112].

By making sure the code containing faults is executed, there is a greater likelihood that the faults will infect the data state and propagate to output. Thus, many program-based adequacy criterion have been developed to determine whether or not a test suite adequately tests the application [118]. A test with high adequacy covers more of the program structure and therefore is more likely to reach parts of the program containing defects.

There are many test adequacy criteria that may be used to evaluate the quality of a test suite. Program-based adequacy criterion involve analysis of the program's structure. For

example, structurally-based metrics may measure how many nodes, branch edges, or paths are visited when the test is run.

Statement coverage criteria and branch coverage criteria are the two most popular examples of structural adequacy criteria that are used in existing coverage analysis tools and by industry [118]. Statement coverage criteria are all-nodes based. Since the fault-failure model indicates that it is impossible to reveal a fault without including that line in a test, all statements in a program need to be executed [112]. A complete path is a path in a control flow graph that starts at the program graph's entry node and ends at its exit node [49]. Then the all-nodes (e.g., statement coverage) test adequacy criterion can be defined as the following by Kapfhammer [49]:

Definition 4 *A test suite T for control flow graph $G = (N, E)$ satisfies the all-nodes test adequacy (statement coverage) criterion if and only if the tests in T create a set of complete paths Π_N that include all $n \in N$ at least once.*

While statement coverage criterion will execute all statements in an application's code, it still may not execute all of the transfers of control within the application's control flow graph. By covering all of the edges in a control flow graph, all nodes will also be covered. A test adequacy criterion C_α is said to subsume a test adequacy criterion C_β if every test suite that satisfies C_α also satisfies C_β [49]. The all-edges test adequacy criterion (e.g., branch coverage) subsumes statement coverage tests, and thus every test suite that satisfies statement coverage also satisfies branch coverage. The branch coverage test adequacy criterion can be defined as the following [49]:

Definition 5 *A test suite T for control flow graph $G = (N, E)$ satisfies the all-edges test adequacy (branch coverage) criterion if and only if the tests in T create a set of complete paths Π_E that include all $e \in E$ at least once.*

In this research, statement and branch coverage are calculated using our THeME system. When using an existing coverage calculating tool called Emma for evaluating Java programs,

we instead look at the forms of coverage criteria that are provided by the tool [91]. These coverage criterion include class coverage, method coverage, and block coverage.

An executable class is considered to have been covered if it has been loaded and initialized, and a method is considered to be covered when it has been entered. A basic block is a sequence of bytecode instructions without any jumps or jump targets. If the basic block is entered, it executes as one atomic unit. Because several source lines can be in the same basic block, including non-executable lines of code such as comments or import statements, it makes sense to keep track of basic blocks rather than individual lines at the time of execution. As branching logic is created in the code, basic blocks are created. Thus, basic block coverage is a very desirable type of coverage metric and it can be obtained with less overhead than other structural criteria such as path coverage. Note that 100% basic block coverage always implies 100% executable line coverage. However, the converse is not true [91].

2.1.2 Measuring Test Suite Effectiveness

A test suite's quality is estimated using the coverage metrics described in Section 2.1.1, but to measure its effectiveness in the presence of errors, more is needed. One method of measuring effectiveness is by the rate of faults detected.

To quantify the goal of increasing a subset of the test suite's rate of fault detection, a metric called $APFD$ developed by Rothermel et al. can be used. The $APFD$ measures the rate of fault detection per percentage of test suite execution [89,90], and it is calculated by taking the weighted average of the number of faults detected during the run of the test suite. $APFD$ can be calculated as follows using a notation introduced by Kapfhammer [49]:

Let T be the test suite under evaluation, g the number of faults contained in the program under test P, n the total number of test cases, and $reveal(i, T)$ the position of the first test in T that exposes fault i.

$$APFD(T, P) = 1 - \frac{\sum_{i=1}^{g} reveal(i, T)}{ng} + \frac{1}{2n}.$$

Faults	Test Cases						
	T_1	T_2	T_3	T_4	T_5	T_6	T_7
f_1	X			X			
f_2			X				
f_3		X					X
f_4				X			
f_5		X				X	

Table 2.1: Faults detected by test suite $T = \langle T_1, \ldots, T_7 \rangle$.

For example, suppose that we have the test suite $T = \langle T_1, \ldots, T_7 \rangle$ and we know that the tests detect faults $f_1, \ldots, f_5$ in P according to Table 2.1. Consider the two prioritized test suites T_1 with test sequence $\langle T_3, T_2, T_1, T_6, T_5, T_4, T_7 \rangle$ and T_2 with test sequence $\langle T_1, T_5, T_2, T_4, T_6, T_7, T_3 \rangle$. Incorporating the data from Table 2.1 into the APFD equation yields

$$\begin{aligned} APFD(T_1, P) &= 1 - \frac{3+1+2+6+2}{7*5} + \frac{1}{2*7} \\ &= 1 - 0.4 + 0.07 \\ &= 0.67, \end{aligned}$$

and

$$\begin{aligned} APFD(T_2, P) &= 1 - \frac{1+7+3+4+3}{7*5} + \frac{1}{2*7} \\ &= 1 - 0.51 + 0.07 \\ &= 0.56. \end{aligned}$$

Thus, according to the $APFD$ metric, T_1 has a better rate of fault detection than T_2, and is therefore more desirable. Note that calculating $APFD$ is only possible when prior knowledge of faults is available. $APFD$ calculations therefore are only used for evaluating test suites after faults are known.

2.2 Related Work

In this section, we discuss related work regarding test case selection and prioritization, structural code coverage evaluation, and leveraging hardware performance monitoring.

2.2.1 Test Selection and Prioritization

The test selection techniques presented in this dissertation are related to existing approaches in both test suite selection and prioritization.

Many test suite selection algorithms (e.g., [16,23,24,41,63,79,87,88,119,120]) have been developed to reduce the number of test cases that must be executed to achieve sufficient fault detection capability, although some of the original test suite's quality may be lost. Minimization-based test selection techniques (e.g., [33,43,61,62,105] attempt to select minimal sets of test cases from a test suite that yield coverage of modified or affected portions of the program. Dataflow-coverage-based regression test selection techniques (e.g., [42,80,104]) select test cases that exercise data interactions that have been affected by modifications. Several safe regression test selection techniques have also been proposed. Techniques that are not safe can fail to select a test case that would have revealed a fault in the modified program [24,58,87,113].

While these techniques are successful in reducing the number of tests that must be run, their selection algorithms are based on estimated fault-finding capability and do not make any guarantees that the selected tests will operate within a specified bound on resources.

There are also many existing approaches to test suite prioritization that focus on the coverage of the structural entities within the program under test. Unlike our time-constrained selection and prioritization technique, none of these prioritization schemes guarantee that the execution of a reordered test suite will terminate within a specified testing time limit, and most rely on knowledge of program modifications or past testing history. Like the present work, Kim and Porter acknowledge that testing often occurs in a time constrained environment [54]. Using methods from statistical quality control, their technique uses test

histories (that include test cost information) to prioritize a test suite according to either code coverage or fault detection. Srivastava and Thiagarajan report on a testing tool that prioritizes a test suite according to the coverage of program changes at the basic block level [102]. Even though their *Echelon* tool can consider the execution time of each test case, the experimental analysis does not evaluate this configuration of their testing framework. Elbaum et al. also present prioritization algorithms that incorporate both the cost and the criticality of a test case [30]. Unlike our approach, their method does not heuristically solve instances of the knapsack problem in order to prioritize test suites for time constrained execution.

Similar to our research, Elbaum et al. and Rothermel et al. focus on general test prioritization and the identification of a single test case reordering that will increase the effectiveness of regression testing over many subsequent changes to the program [29, 89]. Do et al. present an empirical study of the effectiveness of test prioritization in a testing environment that uses JUnit [27]. This paper is also related to our work because Do et al.'s prioritization technique uses coverage information at the method and block levels. Li et al. describe and empirically evaluate a number of test suite prioritization algorithms, including one approach that uses a genetic algorithm and four others that use hill climbing, greedy, additional greedy, and k-optimal greedy methods [65]. Unlike our technique, none of these algorithms take into account a time budget. Finally, Yoo and Harman describe a multi-objective test case selection algorithm that can support time-aware prioritization by identifying how different test orderings balance both cost and coverage [123]. In our problem statement, however, there are no conflicting objectives. We assume that our test cases selection will be allowed to run for its entire allotted time because it was specifically generated to execute under those given constraints. Thus, unlike Yoo and Harman [123], we emphasize the total coverage of a prioritization, letting the speed at which faults are found be a secondary concern.

Recent research by Memon et al. and Karlsson et al. also assumes that the building and testing of a software application is constrained by the amount of time available in an evening

[51,73]. While Memon et al.'s testing infrastructure is highly automated, it does not directly consider the time constraint and thus cannot ensure that testing will always complete in the allotted time [73]. Instead of focusing on testing tools and techniques, Karlsson et al. discuss software process guidelines that govern the routine of a daily build [51]. Finally, Saff et al. explain how to use test factoring to automatically generate fast unit tests from slower system tests [93] and Saff and Ernst explore the use of these factored test cases within a continuous testing methodology [94]. Since this prior research proposes methods for improving the efficiency of testing, they are complementary to the testing algorithms presented in this dissertation.

Finally, Kapfhammer et al. describe a test suite execution infrastructure that can execute tests when memory is constrained and subsequently reduce the time overhead associated with testing [50]. However, this technique does not provide guarantees that testing will operate within a specified bound on time and space overhead.

Several other techniques have been developed that combine test selection and prioritization. Jones and Harrold [48] created a test-suite reduction and prioritization technique that can incorporate aspects of modified condition/decision coverage, which is a multiple entity criteria. In this work, they generate reductions and prioritizations based on properties of modified condition/decision coverage. Smith et al. [98] present an approach that constructs tree-based models of a programs behavior during testing and employs these trees while reordering and reducing a test suite. Their test reduction component identifies a subset of the original tests that covers the same call tree paths. Their prioritization technique reorders a test suite so that it covers the call tree paths more rapidly than the initial test ordering. Smith and Kapfhammer [99] also evaluate four existing algorithms for test suite reduction and prioritization techniques. They enable these techniques to use greedy choice metrics that consider both test case cost and the ratio of coverage to cost.

Each of these techniques focus on reduction in the number of test cases, but they, like the earlier techniques discussed, do not make any guarantees that final test selections can execute within specified resource constraints.

2.2.2 Executing Test Cases Efficiently

There exist a number of tools that can be used to evaluate the structural quality of test cases during program execution. Some of these include CoverageMeter [106], Bullseye-Coverage [25], Clover [14], gcov [], Emma [91], JCover [103], IBMs Rational Suite [46], and Cobertura [2]. A comparison of these tools has been written by Yang et al. [121]. Each of these tools statically instrument a program with probes that remain for the entire execution of the tested program. Instrumentation is also placed along infeasible paths unnecessarily [17]. The time overhead of using instrumentation for monitoring branches with these tools has been reported to be, on average, between 10% to 30%, with code growth ranging from 60% to 90% [76, 95, 106]. When monitoring statement coverage, execution is slowed down by a factor of 7 on average [107].

Several other tools exist for performing dynamic instrumentation for test coverage evaluation. These include PIN [69], Dyninst [44], Paradyn [74], and Jazz [75, 76]. Dyninst and Jazz appear to incur the lowest time and memory overheads when dynamically inserting and removing instrumentation to evaluate test quality. Tikir and Hollingsworth use a dynamic technique for node coverage with Dyninst [44, 107]. The Dyninst tool dynamically inserts instrumentation on method invocations for node coverage. A separate thread periodically removes the instrumentation. This instrumentation remains until collected, even when it is not needed. They report slowdowns of 1.001 to 2.37 (average 1.36) for C programs. Memory needs were not evaluated.

In Jazz, a tool developed by Misurda et al. [75, 76], instrumentation is similarly added only when needed for monitoring. In their tool, however, instrumentation is removed as soon as possible. Misurda et al. reported that for their tool, JAZZ, the slowdown over uninstrumented code for their dynamic approach varies from 0.98 to 1.56 with a 1.18 average slowdown when calculating branch coverage. Memory needs for branch coverage execution are on par with static instrumentation techniques [76].

2.2.3 Hardware Performance Monitoring and Sampling

Instead of evaluating structural quality of test cases through the use of instrumentation, in this dissertation, we exploit hardware performance mechanisms. Much work exists that leverages hardware performance monitoring support for event tracking in optimization, profiling, and debugging, but the potential of leveraging advances in hardware monitoring and multicore technology has been little researched in software testing.

The work by Shye et al. [97] is most closely related to our research regarding using hardware mechanisms for monitoring. Their technique calculates basic block coverage using a combination of static analysis and Branch Trace Buffer (BTB) samples for the purposes of debugging. The BTB, available on the Itanium-2, is much like the LBR in that it is a circular buffer that stores the instruction and target addresses of branches executed. However, the BTB holds only four branches. In their work, after gathering all branch vector information, each vector is mapped to a partial path to calculate basic block coverage. Using this technique, they observe on average 47% of actual number of covered basic blocks using a sampling period of 100K with a performance overhead of approximately 25%. To improve coverage precision, they demonstrate the coverage increase when sampling is supplemented by a dominator analysis. Also, they perform aggregated runs, which is undesirable when testing, to try to gather more complete data.

In this dissertation, we use a more commonly available hardware mechanism that tracks the last sixteen executed branches. This allows for more consecutive branch information to be observed and for more samples to be gathered per period than if using the BTB. Thus, our techniques achieve higher quality coverage data at lower sampling rates. We also implement more sophisticated sampling techniques and use multicore technology to improve the quality of our branch coverage approach. When performing statement coverage, our techniques use simpler hardware mechanisms that are available on nearly all devices, unlike the BTB.

Following the work of Shye et al. [97], Tran et al. [110] use specialized hardware to improve monitoring of branches. Using this hardware, they are able to achieve nearly 100% coverage with only 8% to 12% overhead. However, the hardware used is specialized, and

the benchmarks are not standardized.

Hardware mechanisms have been successfully applied outside of testing and debugging for low overhead profiling of microarchitectural events. While hardware counters have been used in areas such as cache profiling [60], they have also proven useful for path profiling [10]. In work by Azimi et al. [15], a technique to use limited performance counters to simultaneously profile multiple events using sampling for performance analysis is introduced. Recent work by Ramasamy et al. [85] uses retired instruction events to dynamically calculate edge frequency estimates for profiling with a time overhead of less than 2% and no size increase. Mars and Hundt [70] and Chen et al. [22] use hardware performance monitors to aggressively tune dynamic optimizations. Yilmaz and Porter [122] also recently applied hardware mechanisms to distinguish failed executions from successful executions at a fraction of the runtime overhead cost of using software-based execution data. Finally, Sheng et al. [96] created a novel race detection tool that samples memory traces by sampling hardware mechanisms rather than using invasive instrumentation.

Sampling has also been used in software tasks, but without the use of hardware mechanisms, to improve efficiency. In work by Arnold and Ryder [13], instrumentation sampling is used to reduce the overhead of using complete sampling for profile collection. Their framework switches between instrumented and non-instrumented code by placing a sample condition on all method entries and backedges. A sample condition is checked, potentially causing the tool to execute fully instrumented code, based on a trigger mechanism. Using this combination of instrumented and non-instrumented code resulted in above 90% accurate profiles with 6% overhead.

Lightweight instrumentation combined with sampling of program executions has also been used for statistical bug isolation [66, 67]. Although these works do not focus on sampling techniques or applications of hardware, they demonstrate how instrumentation and sampling can be used together to produce highly accurate but low overhead results.

Chapter 3

Knapsack Solvers for Time-Aware Selection

In many development settings that are popular today, there is a set amount of resources budgeted for software testing [3,4,6–8]. All test cases must be executed in a limited amount of time, often just overnight, in order to maintain a high level of program quality and to support rapid development. Current test selection techniques reduce the cost of the original test suite by removing tests that are unlikely to find faults. However, selection techniques only reduce test suites to the point of not degrading the quality of the original test suite. They do not guarantee that the resulting tests will execute within an particular set of constraints. Prioritization techniques may take constraints into account, but they still make the assumption that all tests in the test suite can be executed. If testing is halted early due to constraints, test prioritization techniques aim to ensure that the tests that are most likely to find bugs will execute as early as possible.

In this chapter, we discuss how the selection and prioritization problems can be combined to produce high quality test selections that are guaranteed to execute within given resource constraints. The resource-constrained test case selection problem is similar to the NP-complete zero/one knapsack problem [36, 89]. The 0/1 knapsack problem can be described in the following manner: given a knapsack with fixed capacity and a set of distinct items each with its own value and weight, find the maximum cumulative value of items that can fit in the knapsack such that the sum of the item weights in the knapsack does not exceed

the knapsack's capacity [52].

We first present a description of the resource-aware test suite challenge as a 0/1 knapsack problem. We then describe seven knapsack approximation algorithms that can be used for test case selection within resource constrained environments. The goal is to produce a test suite selection that is guaranteed to always execute within given resource constraints. Additionally, the selection should have the a high potential for overall fault detection compared to the original test suite. Finally, the ordering of the test cases within the selection should be likely to find faults earlier in test execution rather than later. By having the highest quality tests execute early, we can provide faster feedback on the system under test.

In our experimental analysis, we empirically evaluate the efficiency and effectiveness of the seven techniques that construct resource-aware test suite selections. When selecting test cases, we constrain the amount of time available for testing. Power consumption and monetary budgets are analogous constraints and could be substituted for time in each algorithm.

In lieu of directly measuring fault detection effectiveness, we analyze three metrics. The first is code coverage, described in Section 2.1.1. Code coverage is the percentage of the structural elements within the program (e.g., basic blocks or methods) that are executed during testing, with a high value indicating strong effectiveness. The second metric, coverage preservation, is the proportion of code covered by the selected tests compared to the code coverege of the original test suite. We also define a third metric, which we call order-aware coverage. Order-aware coverage gives preference to test selections that cover a greater amount of code earlier in the execution phase of the selected tests. A higher order-aware coverage implies that the test cases selected are also ordered in a way to be more likely to find faults early in test execution rather than later.

We experimentally determine that if there is little overlap between the test cases, greedy approaches to the test selection problem are particularly effective and require the least time and memory. More sophisticated selection techniques are likely to maintain a higher level of quality than simple solvers, although neither group can make any guarantee regarding

the final cumulative coverage of the result.

3.1 Time-Aware Selection

To address the problem of constructing a time-constrained test suite selection, we use traditional techniques to solve the 0/1 knapsack problem. We define a test suite T as described by Definition 1. The selection of T is denoted T'. The test cases within T are written as defined in Definition 2. As is preferred in testing practice, we require that each test in T be independent (Definition 3) so that we can guarantee that for all $T_i \in \langle T_1, \ldots, T_n \rangle, \Delta_{i-1} = \Delta_0$ [49,82]. Thus there are no test execution ordering dependencies. This requirement enables the selection technique to select and reorder tests into any sequence that maximizes the suite's ability to isolate defects.

In the context of the 0/1 knapsack problem, the maximum amount of time within which a selected test suite must run is the maximum capacity of the knapsack, the test cases are the knapsack items, each test case's execution time is its weight, and its percentage of code coverage is its value. When these values are passed into a 0/1 knapsack algorithm, the output is a final solved knapsack, namely, a test case selection that fits within the desired time limit.

As defined by Kellerer et al., the 0/1 knapsack problem can be defined formally in terms of test suite selection in the following manner [52]:

$$\begin{aligned} &\text{Maximize:} && \textstyle\sum_{i=1}^{n} c_i x_i \\ &\text{Subject to:} && \textstyle\sum_{i=1}^{n} t_i x_i \leq t_{max},\ x_i = 0 \text{ or } 1, \end{aligned}$$

where c_i is the code coverage, t_i is the execution time of test case T_i, and t_{max} is the maximum time allowed for the execution of the selection.

3.2 Knapsack Solvers as Selectors

The 0/1 knapsack problem is an NP-complete problem [36]. There are a number of algorithms that approximate the optimal solution to this problem, which vary in complexity

and optimality. Seven knapsack algorithms are used in this chapter and are described in terms of the test suite selection problem as follows:

Random: While the total execution time of the selection is less than or equal to the maximum allowed time limit, select a test case T_i randomly from the set of unused test cases and add it to the selection. If the addition of a test case causes the maximum time limit t_{max} to be exceeded, remove that test case and return the remaining test case ordering.

Greedy by Ratio: For each test case T_i, calculate the code-coverage-to-execution-time ratio, $\frac{c_i}{t_i}$. Sort the test cases in descending order according to this ratio, then successively place test cases from this ordering into the selection until the addition of the next test would cause the maximum time limit to be exceeded. **Greedy by Value** and **Greedy by Weight** are performed similarly, except code coverage and test execution time are used in place of the code-coverage-to-execution-time ratio, respectively.

Dynamic Programming: Divide the problem into subproblems, and solve each piece separately, storing the answers so as to avoid repeatedly solving the same problem [52]. The total code coverage of the selection for i test cases and t_{max} execution time is zero if there are no test cases in the solution or if t_{max} is zero. Otherwise, the solution for i test cases and t_{max} time either includes the ith test case or does not. In the first case, the total code coverage of the selection for i test cases and t_{max} time is equal to the total code coverage of the selection for $i-1$ test cases and $t_{max}-t_i$ time plus the code coverage of the ith test case, c_i. In the second case, the total code coverage is equal to the code coverage of the selection for $i-1$ test cases and t_{max} time. The best solution is chosen as the final test suite selection.

Generalized Tabular: Like dynamic programming, solve subproblems of the main problem using a large table [39]. The table has $t_{max}+1$ rows and $n+1$ columns, where n is the number of test cases. Each row i represents a problem with maximum time limit ta_i for values 0 to t_{max}. The last item in each row is the optimal coverage solution, denoted $copt_{ta_i}$, for that problem. An example with test cases $\langle T_1, T_2, T_3 \rangle$ having coverages 2, 1, 3, and times 3, 4, 5, respectively, and $t_{max} = 5$ is shown in Figure 3.1. As seen in the example,

ta	T_1	T_2	T_3	$copt$
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	2+$copt_0$	0	0	2
4	2+$copt_1$	1+$copt_0$	0	2
5	2+$copt_2$	1+$copt_1$	3+$copt_0$	3

Figure 3.1: Generalized Tabular Example.

the optimal coverage for each time limit is stored in the last column of the table, while the rows are numbered by the time limits. The other columns each correspond to a test case T_j. Element a_{ta_i,T_j}, as shown in Figure 3.1, is equal to $c_j + copt_{ta_i - t_j}$ if T_i can be added within the time limit ta_i, and 0 otherwise. After the table is complete, the full solution is recovered by working backward and retracing the steps taken to compute the elements of the table.

Core: Create a "core" solution using a subset of test cases, then use this core to find a solution to the overall selection problem [83]. First, find a good solution (the core solution) using the greedy by weight algorithm. Then, using the dynamic programming algorithm, try to find a better solution by replacing each test case in the core solution with another unused test case.

In addition to these seven techniques, each algorithm can be used in conjunction with *scaling*. When scaling, the problem is reduced by means of a theorem described by Gossett [39]. The version specific to the selection problem addressed in this chapter follows.

Suppose that for a selection with maximum allowed execution time t_{max}, there are n test cases in the test suite. Denote the code coverage values of the test cases by $c_1, c_2, \ldots, c_n$ and the execution time of the test cases by $t_1, t_2, ..., t_n$. Assume the test cases have already been ordered such that $\frac{c_1}{t_1} \geq \frac{c_2}{c_2} \geq \ldots \geq \frac{c_n}{t_n}$. If $c_1 \times \left\lfloor \frac{t_{max}}{t_1} \right\rfloor \geq c_2 \times \left(\frac{t_{max}}{t_2} \right)$, then it is possible to find an optimal knapsack solution that includes T_1.

To perform scaling, order all test cases by their code-coverage-to-execution-time ratios, as indicated by the theorem. Check if the inequality $c_1 \times \left\lfloor \frac{t_{max}}{t_1} \right\rfloor \geq c_2 \times \left(\frac{t_{max}}{t_2} \right)$ holds. If it

Test Case	T_1	T_2	T_3	T_4	T_5	T_6
coverage	4	5	2	6	8	1
time	105	60	60	95	225	32
c/t ratio	0.0381	0.0833	0.0333	0.0632	0.0356	0.0313

Figure 3.2: Example Test Cases.

Comparison	**Inequality**	**Action**
Compare T_2 and T_4	$5 \times \lfloor \frac{445}{60} \rfloor \geq 6 \times (\frac{445}{95})$ $35 \geq 28.1053$	Add T_2 to T' Time left = 385
Compare T_4 and T_1	$6 \times \lfloor \frac{385}{95} \rfloor \geq 4 \times (\frac{385}{105})$ $24 \geq 14.6667$	Add T_4 to T' Time left = 290
Compare T_1 and T_5	$4 \times \lfloor \frac{290}{105} \rfloor \geq 8 \times (\frac{290}{225})$ $8 \not\geq 10.3331$	Not conclusive Time left = 290

Figure 3.3: Scaling Heuristic Example.

does, put T_1 in the selection and subtract the execution time t_1 of T_1 from the maximum execution time t_{max}. Now consider the selection with maximum execution time $t_{max} - t_1$ for the remaining list of test cases, with T_2 now occurring first in the list, and so on. Continue down the list in this manner until the inequality ceases to hold—let us say that this occurs at T_i—and then stop. The $i - 1$ test cases placed in the knapsack through this process are guaranteed to be part of an optimal solution for the selection with maximum allowed execution time t_{max}. Finish by using any of the aforementioned techniques on the remaining unselected test cases. These test cases have maximum allowed execution time $t_{max} - \sum_{j=1}^{i-1} t_j$. The test cases that will be in the final solution for the selection with maximum allowed execution time t_{max} will be those in the solution for the selection with maximum allowed execution time $t_{max} - \sum_{j=1}^{i-1} t_j$ plus those determined to be part of the optimal solution by the scaling heuristic.

For example, suppose there are six test cases, with code coverages and execution times as shown in Figure 3.2, and a maximum execution time of 445 units. First, the test cases are ordered according to their code-coverage-to-execution-time ratio to yield

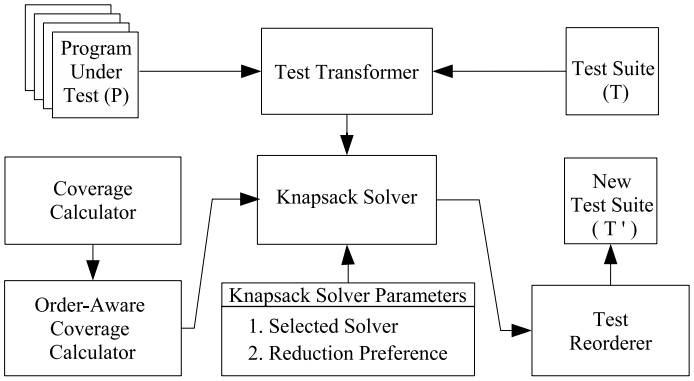


Figure 3.4: Overview of the Selection Infrastructure.

$\langle T_2, T_4, T_1, T_5, T_3, T_6 \rangle$. Then comparisons are performed, as shown in Figure 3.3. After the last comparison fails, the heuristic no longer yields any information, so the rest of the problem is solved using one of the seven algorithms described in this section.

3.3 Experiment Goals and Design

The goals of our experiments are to:

1. Measure empirically, using two case studies, the efficiency of seven knapsack algorithms used in prioritization, each with and without the use of a scaling heuristic, in terms of time and memory overhead.

2. Record, graph and analyze the effectiveness of each of these algorithms with and without scaling in terms of three coverage-based metrics: code coverage, coverage preservation, and order-aware coverage.

All of the algorithms described in this chapter were implemented in Java and were used to prioritize JUnit test cases from two case study applications, described below. The prioritizations were performed on a dual-core AMD Opteron Processor, each core being 1.8 GHz, running the Fedora Core 3 GNU/Linux operating system with 2 GB of main memory and 2048 MB maximum heap size. To perform a prioritization, first the execution time

	`Gradebook`	`JDepend`
Classes	5	22
Functions	73	305
NCSS	591	1808
Test Cases	28	53
Test Exec. Time	7.008 s	5.468 s

Figure 3.5: Case Study Applications.

and code coverage information of each test case in the test suite is recorded. From this information, a set of knapsack items is created. Next these items are used as input to the knapsack algorithms. Each algorithm returns a list of test cases representing the final test suite prioritization, as depicted in Figure 3.4. As the algorithms run, time overhead and memory information is gathered. Afterwards, code coverage, coverage preservation, and order-aware coverage information are calculated for each test suite selection. The time, memory, and coverage information is used to compare the algorithms and examine the key trade-offs.

3.3.1 Case Studies

In order to measure the effectiveness of these algorithms in test suite selection, the test suites of two case study programs were used, `JDepend` and `GradeBook`, both of which have independent test suites and different characteristics. These test suites were manually implemented by the developers of the tools. `JDepend` is a tool for creating design quality metrics for Java packages in terms of extensibility, reusability, and maintainability. `GradeBook` is a program that provides functions to perform tasks associated with creating and maintaining a grade book system for a course. Figure 3.5 gives information regarding each application and their test suites. The test cases of `Gradebook` differ from those in `JDepend` in that they are I/O-bound by their frequent interactions with a database. On average, `Gradebook`'s test cases take longer to run, while `JDepend`'s test cases have very short execution times.

3.3.2 Evaluation Metrics

In order to measure the effectiveness of these algorithms, three metrics were used: code coverage, coverage preservation, and order-aware coverage. These were used despite the fact that ideally, the effectiveness of a test suite selection would be based on the average percentage of faults it detected given a time constraint, described in Section 2.1.2. However, since the nature and location of faults are unknown and unique to each program, it is not possible to calculate the average percent of faults detected, unless faults are artificially seeded into a program. While this can be a useful way of empirically judging the effectiveness of a selection, it runs the risk of not being representative of the type and number of faults that occur in real-world applications. Therefore, coverage information, which has been shown to be highly correlated with fault-detection potential, is used [45, 112, 115, 118].

Code coverage, denoted $cc(P, T')$, where P is the program being tested, is a measure of the percentage of program source statements that are executed when the selected test suite is run. There are several different levels of granularity at which code coverage can be measured; in this chapter, we use block coverage, which is described in Section 2.1.1

Coverage preservation, denoted $cp(P, T, T')$, is a proportional measure of the amount of code covered by the time-aware selection versus the amount of code covered by the entire test suite. In other words,

$$cp(P, T, T') = \frac{cc(P, T')}{cc(P, T)} \tag{3.1}$$

Order-aware coverage, denoted $\mathcal{C}(P, T')$, takes into account not only the percentage of code covered by test cases in a selection, but also the order in which the test cases in the selection execute. This provides a way to measure the amount of code covered in conjunction with the time during the execution phase at which that code was covered. A high order-aware coverage score implies that i) the overall coverage of the test case selection is high and ii) more of the source code is covered by test cases that are executed early in the test selection. This second implication is important because it is desirable to execute test cases with the highest fault-detecting potential earlier in the test suite execution phase.

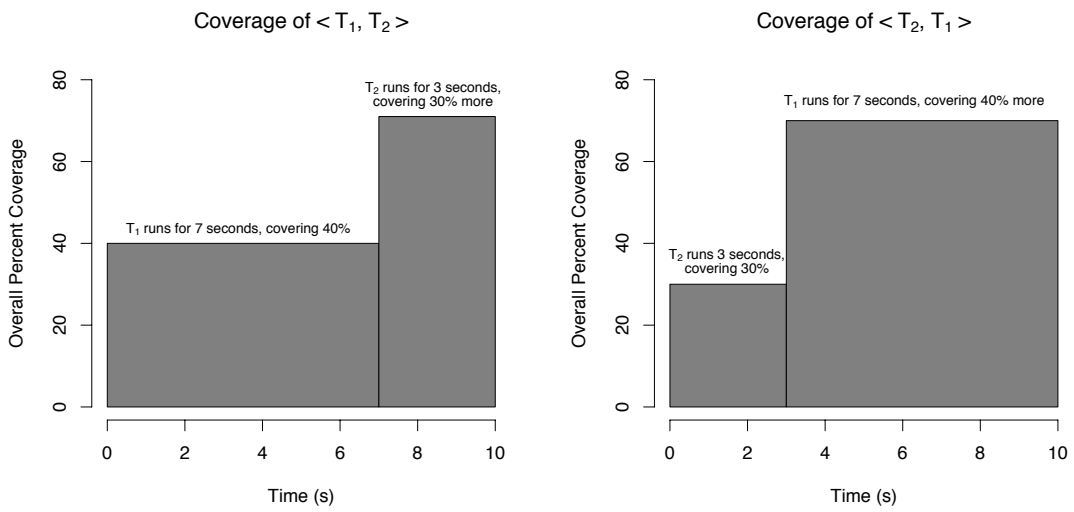


Figure 3.6: $\langle T_2, T_1 \rangle$ covers more requirements early in execution than $\langle T_1, T_2 \rangle$.

Order-aware coverage is calculated in two parts, primary and secondary. Let T' be a selection of test cases of T for program P. Because we assume that the entire selection will be executed, we give primary consideration to the amount of code covered by the test cases selected when calculating the order-aware coverage. The earliness of coverage within the prioritization is a secondary concern and should be considered separately. Thus, we are not attempting to find a good trade-off between the two objectives. Rather, our metric determines the best selections based only on coverage and then breaks ties based on when code is covered during test suite execution. It is then appropriate to divide order-aware coverage, $\mathcal{C}$ into two parts such that $\mathcal{C}(P, T') = \mathcal{C}_{pri}(P, T') + \mathcal{C}_{sec}(P, T')$, where $\mathcal{C}_{pri} \in [0, 100]$ considers the first criteria and $\mathcal{C}_{sec} \in [0, 1]$ deals with the second.

For example, consider a test suite with test cases T_1, T_2, and T_3 where T_1 covers 40% of requirements in 7 seconds, T_2 covers 30% in 3 seconds, and T_3 covers 10% in 2 seconds. For simplicity, assume that T_1, T_2, and T_3 cover disjoint requirements. Let the time budget $t_{max} = 10$ seconds. Then possible selections include $\langle T_1, T_2 \rangle, \langle T_2, T_1 \rangle, \langle T_1, T_3 \rangle, \langle T_3, T_1 \rangle, \langle T_2, T_3 \rangle$, and $\langle T_3, T_2 \rangle$. So, $\langle T_1, T_3 \rangle, \langle T_3, T_1 \rangle, \langle T_2, T_3 \rangle$, and $\langle T_3, T_2 \rangle$ all cover fewer requirements within the time budget than $\langle T_1, T_2 \rangle$ and $\langle T_2, T_1 \rangle$, which cover 70%. Thus, these are less preferable than $\langle T_1, T_2 \rangle$ and $\langle T_2, T_1 \rangle$. Note that

$\langle T_1, T_2 \rangle$ and $\langle T_2, T_1 \rangle$ cover the same amount of requirements. Yet, we prefer $\langle T_2, T_1 \rangle$ because executing T_2 first results in covering more requirements earlier in time than if T_1 were run first, as is evident in Figure 3.6. The calculation of the actual order-aware coverage metric values for this example are given below.

The first component of order-aware coverage, $\mathcal{C}_{pri}$ is calculated by measuring the code coverage cc of the entire test selection T'. The coverage cc is a percentage between 0 and 100.

The second component $\mathcal{C}_{sec}$ considers the incremental code coverage of the selection, giving preference to test selections whose earlier tests have greater coverage. Note again that $\mathcal{C}_{pri} \in [0, 100]$ is the more important value and represents the percentage of requirements covered by the selection within the time limit. $\mathcal{C}_{sec} \in [0, 1]$ then breaks ties between selections covering equal code amounts, changing only the mantissa of the order-aware score.

$\mathcal{C}_{sec}$ is also calculated in two parts. First, $\mathcal{C}_{s-actual}$ is computed by summing the products of the execution time $time(\langle T_i \rangle)$ and the code coverage cc of $T'_{\{1,i\}} = \langle T_1 \dots T_i \rangle$ for each test case $T_i \in T'$.

$$\mathcal{C}_{s-actual}(P, T') = \sum_{i=1}^{|T'|} time(\langle T_i \rangle) \times cc(P, T'_{\{1,i\}}) \tag{3.2}$$

$\mathcal{C}_{s-max}$ represents the maximum value that $\mathcal{C}_{s-actual}$ could take (i.e., the value of $\mathcal{C}_{s-actual}$ if T_1 covered 100% of the code covered by T'.) For a T',

$$\mathcal{C}_{s-max}(P, T') = cc(P, T') \times \sum_{i=1}^{|T'|} time(\langle T_i \rangle) \tag{3.3}$$

Finally, $\mathcal{C}_{s-actual}$ and $\mathcal{C}_{s-max}$ are used to calculate the part of the order-aware coverage metric, $\mathcal{C}_{sec}$. Specifically,

$$\mathcal{C}_{sec}(P, T') = \frac{\mathcal{C}_{s-actual}(P, T')}{\mathcal{C}_{s-max}(P, T')} \tag{3.4}$$

As an example of the order-aware coverage calculation, let P be a program. Suppose $T' = \langle T_1, T_2 \rangle$ and $T'' = \langle T_2, T_1 \rangle$. As in the example above, and as shown in Figure 3.6, assume that the test cases of T' have execution times of $time(\langle T_1 \rangle) = 7$ and $time(\langle T_2 \rangle) = 3$,

and code coverage $cc(P, T') = cc(P, T'') = 70$ where the code covered by T1 and T2 is disjoint. Then,

$$\mathcal{C}_{pri}(P, T') = \mathcal{C}_{pri}(P, T'') = 70.$$

$\mathcal{C}_{sec}$ next gives preference to test selections that have more code covered early in execution. To calculate $\mathcal{C}_{sec}$ for T', the code coverages of $T'_{\{1,1\}} = \langle T_1 \rangle$ and $T'_{\{1,2\}} = \langle T_1, T_2 \rangle$ must each be measured. For T'', the code coverages of $T''_{\{1,1\}} = \langle T_2 \rangle$ and $T''_{\{1,2\}} = \langle T_2, T_1 \rangle$ are calculated. Then $cc(P, T'_{\{1,1\}}) = 0.4$, $cc(P, T'_{\{1,2\}}) = 0.7$, $cc(P, T''_{\{1,1\}}) = 0.3$, and $cc(P, T''_{\{1,2\}}) = 0.7$. $\mathcal{C}_{sec}$ is calculated as follows for T' and T'',

$$\begin{aligned} \mathcal{C}_{s-actual}(P, T') &= (7 \times 0.4) + (3 \times 0.7) = 4.9 \\ \mathcal{C}_{s-max}(P, T') &= 0.7(7 + 3) = 7 \\ \mathcal{C}_{sec}(P, T') &= \frac{4.9}{7} = 0.7 \end{aligned}$$

and

$$\begin{aligned} \mathcal{C}_{s-actual}(P, T'') &= (3 \times 0.3) + (7 \times 0.7) = 5.8 \\ \mathcal{C}_{s-max}(P, T'') &= 0.7(7 + 3) = 7 \\ \mathcal{C}_{sec}(P, T'') &= \frac{5.8}{7} = 0.829 \end{aligned}$$

Adding $\mathcal{C}_{pri}$ and $\mathcal{C}_{sec}$ gives the total order-aware coverage value. In this example,

$$\mathcal{C}(P, T') = \mathcal{C}_{pri}(P, T') + \mathcal{C}_{sec}(P, T') = 70 + 0.7 = 70.7$$

and

$$\mathcal{C}(P, T'') = \mathcal{C}_{pri}(P, T'') + \mathcal{C}_{sec}(P, T'') = 70 + 0.829 = 70.829$$

Therefore, although T' and T'' cover the same amount of requirements within the time bud-

get, T'' is preferred because it is more likely to find faults earlier in the selection execution.

All of the coverage information is obtained using Emma, an open source Java code coverage tool that reports code coverage statistics at method, class, package, and all-classes levels [91]. The results reported in this chapter are based on block level coverage, because the use of block level coverage generally gives better results than levels of a coarser grain, such as method level [91, 118]. As this work also examines the trade-offs between the effectiveness of a test suite selection and the time and space overhead incurred in performing the selection, execution time and memory statistics were also obtained. To do so, a Linux process tool, which calculates the peak memory use and total user and system time required by a program, was used.

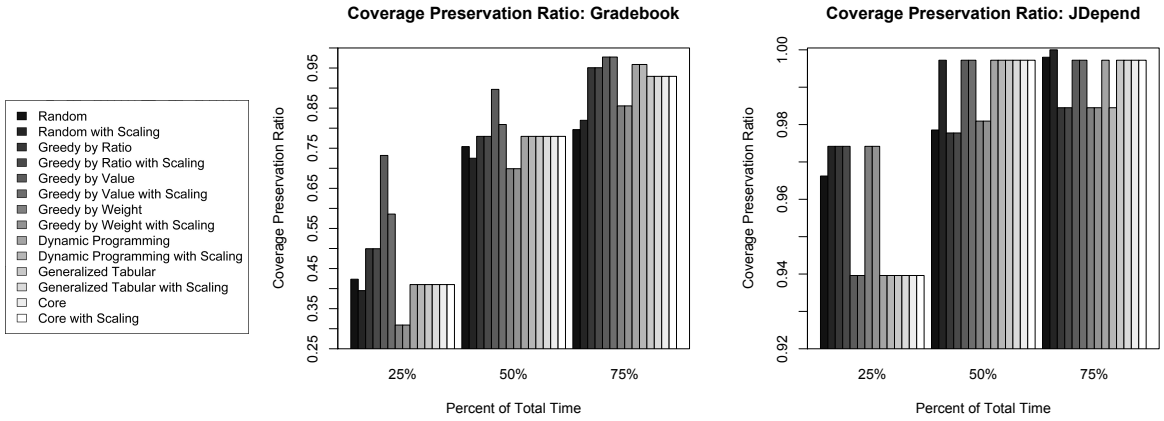
3.4 Experiments and Results

Experiments were run in order to analyze the effectiveness and efficiency of the seven test suite selectors described in Section 3.2. The solvers selected the test suites of `Gradebook` and `JDepend` so that resulting test tuples would execute within 25, 50, and 75% of the total execution time of the initial test suites.

3.4.1 Selection Effectiveness.

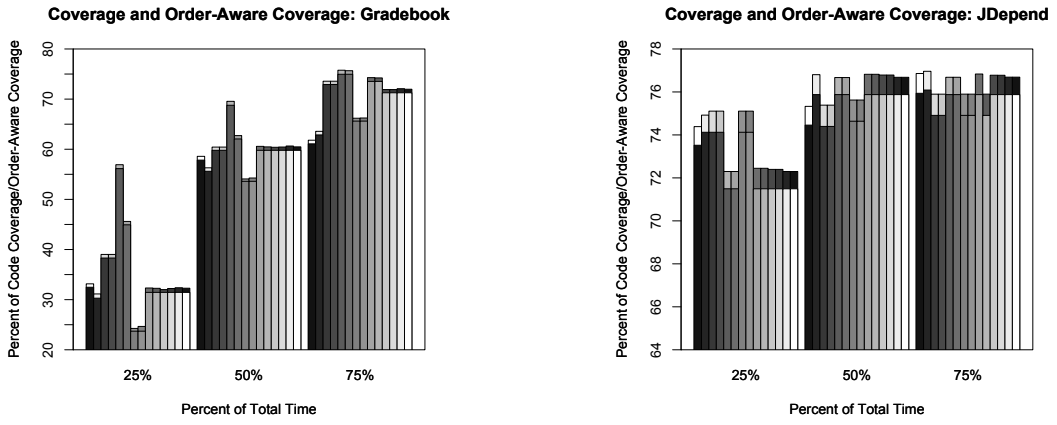
Experiments were run in order to analyze the effectiveness and efficiency of the seven test suite selectors described in Section 3.2. The solvers selected test cases from the test suites of `Gradebook` and `JDepend` so that resulting test tuples would execute within 25, 50, and 75% of the total execution time of the initial test suites.

First, we examine the overall coverage, order-aware coverage, and coverage preservation of each of the resulting selections. These can be seen in Figures 3.7(a) and (b) and 3.8(a) and (b). In Figures 3.7(a) and 3.8(a), we see that greedy by value, solver 3, achieves the highest overall coverage for each testing time constraint for `Gradebook`. Greedy by ratio and greedy by weight, solvers 2 and 4, create the best prioritizations for `JDepend`, as shown



(a) (b)

Figure 3.7: Coverage Preservation of Test Suite Selection.



(a) (b)

Figure 3.8: Overall Coverage and Order-Aware Coverage of Test Suite Selection.

in Figures 3.7(b) and 3.8(b). However, all of the selections made for JDepend maintain more than 93% of the original test suite's coverage, even in the smallest time constraint.

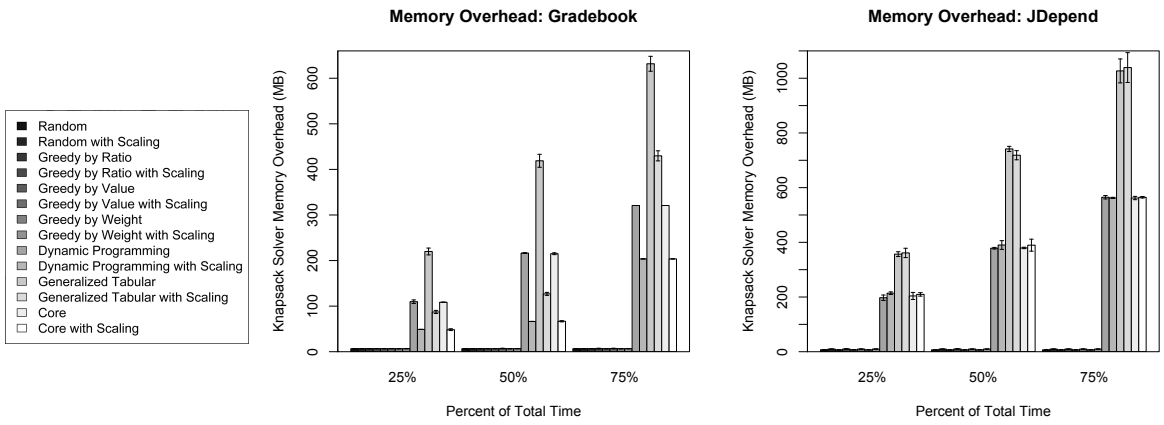
The success of these solvers is understandable in light of the nature of the test suites. In the `Gradebook` test suite, there is only a little coverage overlap between test cases, so a greedy by value approach is likely to add worthwhile test cases to the prioritization at each iteration, which is shown in Figure 3.8(a). `JDepend`'s test cases have very short execution times, and many of them cover about the same amount of code. Thus, a solver that orders the test cases so that the shortest tests run first does well. For such a test suite, a greedy algorithm prioritizing based on the ratio of code coverage to execution time performs equally well, as seen in Figures 3.7(b) and 3.8(b). Note that because the execution time difference between `JDepend`'s test cases is much smaller than that of `Gradebook`'s test cases, we observe a less drastic coverage difference over the JDepend test cases and as the time limit increases. Similar trends are observed between coverage preservation, coverage, and order-aware coverage, which are presented in Figures 3.7 and 3.8.

One might think that the core algorithm would produce best results amongst the solvers. However, in Figures 3.7 and 3.8, we observe that this is not true for either `JDepend` or `Gradebook`. While the core algorithm achieves a higher utility result than other solvers, there is no guarantee that the total coverage will also be high once the overall coverage is considered.

3.4.2 Selection Efficiency.

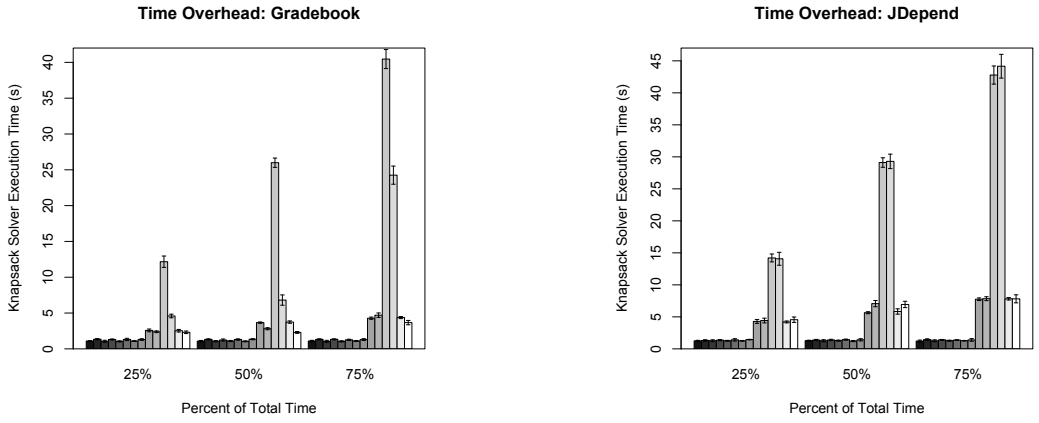
Next we evaluated the time and space overheads incurred by each test selection algorithm, which are displayed in Figures 3.9 and 3.10. Among the knapsack solvers, the time and memory costs were insignificant in all but the dynamic programming, generalized tabular, and core algorithms. In Figure 3.9(b), we see that the memory requirements of the generalized tabular solver were especially prohibitive, reaching over 1039MB at peak usage.

For the seven algorithms described in this chapter, the scaling technique successfully reduced the prioritization execution time. In one case in `Gradebook`, the time was decreased



(a) (b)

Figure 3.9: Memory Overhead of Test Suite Selection.



(a) (b)

Figure 3.10: Time Overhead of Test Suite Selection.

by 330%, as seen in Figure 3.10(a). However, as described in Figure 3.10(b), scaling does not always improve the time overhead. It also occasionally had a negative impact on memory overhead, particularly for `JDepend` in Figure 3.9(b).

3.5 Conclusions

Results indicate that a trade-off must be made between efficiency and final coverage. The design of the test suite is also of great importance. As shown in Figure 3.7(a), if there is little overlap between the test cases, a cheaper prioritizer can be used with favorable results. While more sophisticated solvers such as dynamic programming, generalized tabular, and core are likely to obtain higher utility than simple solvers, neither group can make any guarantee regarding final cumulative coverage of the result. It is therefore likely that for test suites containing a large amount of overlap between test cases, an overlap-aware solver will be advantageous. In the next chapter, we develop an overlap-aware test suite selection technique to create more effective test case selections.

Chapter 4

A Genetic Algorithm for Time-Aware Selection

In this chapter, we present an innovative technique that selects test cases from a test suite for execution within a time-constrained environment through the use of a genetic algorithm. It is our goal that our final resulting test selections, like those in Chapter 3, will satisfy three properties. The test selection i) will always execute within the resource constraints, ii) will have the highest possible potential for overall fault detection, and iii) are likely to find faults earlier in test execution rather than later. Our technique differs from the techniques presented in Chapter 3 in that our genetic algorithm also takes coverage overlap between test cases into account. For example, if test cases T_1 and T_2 each cover the same code, a selection including T_1 and T_2 would have no greater value than a prioritization including either T_1 or T_2. This requires us to define an extended version of the 0/1 knapsack problem in which test cases interact within the constraints.

We first describe the general design of a genetic algorithm and the challenges associated with using a genetic algorithm for test selection in time-constrained environments. We then describe our genetic algorithm technique for solving the test selection problem and provide an example of the genetic algorithm in action.

This chapter provides empirical evidence that the produced prioritizations on average have significantly higher fault detection rates than random or more simplistic prioritizations. In our study, we prioritize the JUnit test suites of two applications that were seeded with a

wide variety of faults. For each application, we considered eighteen different configurations of the genetic algorithm, varying the number of generations, population size, time budget, and test adequacy criterion.

Our experimental analysis of the GA-produced selections with regard to (i) original coverage preservation, (ii) overall coverage, and (iii) fault finding ability demonstrates that the resulting selections lose little of the original test suite's value. Our empirical evaluation reveals the effectiveness of the resulting selections in relation to (i) GA-produced selections using different parameters, (ii) the initial test suite ordering, (iii) the reverse of the initial test suite ordering, (iv) random test suite prioritizations, and (v) fault-aware prioritizations We show that the GA-produced selections are superior to the other test suite reorderings. Finally, our experimental study of performance reveals that our GA-selection technique is most applicable when (i) there is a fixed set of time constraints, (ii) prioritization occurs infrequently, or (iii) the time constraint is particularly small.

4.1 Genetic Algorithms and the Test Selection Challenge

A genetic algorithm (GA) is a search heuristic that mimics the process of natural evolution. This heuristic is often used to find solutions to optimization and search problems, and it is another approach to estimate a solution for the 0/1 knapsack problem.

4.1.1 Designing a Genetic Algorithm

In a genetic algorithm, a population $P = \{c_1, \ldots, c_m\}$ of individuals is randomly created. Individuals are encodings of candidate solutions to an optimization problem. Once a population is created, every individual in the population is evaluated and assigned a fitness value. The fitness function assigns a value to each individual representing its quality. Next, multiple individuals are stochastically selected from the current population based on their fitness and modified to form a new population. If an individual is strong, its genes are more likely to be selected for inclusion in the new population. Selection of individuals for reproduction

Algorithm PROCEDURE GA
(∗ A simple genetic algorithm ∗)
Input: Test suite T;
Size of population m;
Maximum iterations d;
Crossover probability p_C;
Mutation probability p_M;
Output: Final Population P_{t-1}, $0 < t - 1 \leq d$
1. Create a randomly generated initial population, P_0 of m candidate solutions from the set of test cases $\{T_1, \ldots, T_n\}$ in T;
2. Set $t = 0$; **repeat**
3. Calculate the fitness of each $c_j \in P_t$ where $P_t \in \{P_0, \ldots, P_d\}$;
4. **repeat**
5. Select two parent individuals $c_k, c_l \in P_t$, the probability of selection being an increasing function of fitness;
6. Apply crossover to c_k and c_l to form two offspring according to p_C;
7. Apply mutation according to p_M;
8. Place the resulting chromosomes in the new population, P_{t+1};
9. **until** P_{t+1} has size m;
10. $t \leftarrow t + 1$
11. **until** $t > d$ (or reach other termination condition);
12. **return** P_{t-1}

Figure 4.1: Genetic Algorithm Procedure.

is implemented by eliminating the low-fitness individuals, and inheritance is implemented by making copies of high-fitness individuals. After pairs are selected, for example c_k, c_l, the chromosomes are either retained as is, combined using a crossover operation based on the crossover probability p_C, or are mutated by flipping elements of the individual in a probabilistic manner based on the mutation probability p_m. Reproduction continues until the number of individuals in the original population, m, is reached for the new population [77]. The new population is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population [34, 37, 125].

A typical structure for a genetic algorithm is shown in Figure 4.1 [77].

4.1.2 Genetic Algorithm Challenges

There are many challenges to developing a genetic algorithm to solve a particular problem. The first challenge is in determining how to encode an individual, necessary on line 1 of Figure 4.1. Traditionally, individuals are represented in binary as strings of 0s and 1s, but other encodings are also possible.

Determining an appropriate fitness function is one of the most difficult tasks in designing a genetic algorithm. The fitness function calculation is shown on line 3 of Figure 4.1. The fitness function determines how each individual will be interpreted, and it should quantify the optimality of a solution in a genetic algorithm in order to allow that particular individual to be ranked against all the other individuals. Optimal individuals, or at least individuals which are more optimal, are allowed to breed and mix their datasets by any of several techniques, producing a new generation that will be even better [34]. Another way of looking at fitness functions is in terms of a fitness landscape, which is a representation of the space of all possible individuals along with their fitnesses [77]. An ideal fitness function correlates closely with the algorithm's goal, and yet may be computed quickly. Speed of execution is very important, as a typical genetic algorithm must be iterated many, many times in order to produce a useable result for a non-trivial problem. The chief bottleneck in a genetic algorithm is generally in the fitness function calculation [77].

The final challenge is in deciding how the individuals of the population will reproduce. During each successive generation, a proportion of the existing population is selected to breed a new generation. Individual solutions are selected through a fitness-based process, where fitter solutions are typically more likely to be selected, as noted on line 5 of Figure 4.1. Certain selection methods rate the fitness of each solution and preferentially select the best solutions. Other methods rate only a random sample of the population. Once individuals are selected, they are acted upon by genetic operators such as mutation and recombination, shown on lines 6 and 7 of Figure 4.1. The likelihood of mutation and ways of combining are generally determined based on random numbers.

4.1.3 The Test Selection Challenge

Problem 1 defines the time-constrained test case selection problem in terms of a 0/1 Knapsack to be solved with a genetic algorithm. Intuitively, a test tuple, or individual, σ earns a better fitness score if it has a greater potential for fault detection and can execute within the user specified time budget.

Problem 1 *(***Time Constrained Test Suite Selection***)*

Given: *(i) A test suite, T, (ii) the collection of all permutations of elements of the power set of permutations of T, $perms(2^T)$, (iii) the time budget, t_{max}, and (iv) two functions from $perms(2^T)$ to the real numbers, time and fit.*

Goal: *Find the test tuple $\sigma_{max} \in perms(2^T)$ such that $time(\sigma_{max}) \leq t_{max}$ and $\forall \sigma' \in perms(2^T)$ where $\sigma_{max} \neq \sigma'$ and $time(\sigma') \leq t_{max}$, $fit(\sigma_{max}) > fit(\sigma')$.*

In Problem 1, $perms(2^T)$ represents the set of all possible tuples of T. Each individual is selected from the set $perms(2^T)$. When the function *time* is applied to any of these tuples, it yields the execution time of that tuple. The function fit is applied to any such tuple and returns a fitness value for that ordering. Without loss of generality, we assume that a higher fitness is preferable to a lower fitness. In this work, the function fit quantifies a test tuple's incremental rate of fault detection and takes code coverage overlap into account. Our technique considers the potential for fault detection and the time overhead of each test case in order to evaluate whether the test suite achieves its potential at the fastest rate possible.

4.2 Time-Constrained Selection

Our selection technique uses both testing time and potential fault detection information to intelligently select and reorder a test suite that adheres to Definition 1. As is preferred in testing practice, we require that each test in T be independent so that we can guarantee that for all $T_i \in \langle T_1, \ldots, T_n \rangle, \Delta_{i-1} = \Delta_0$ [49, 82]. Thus there are no test execution ordering

Individual Representation	Variable length tuple of test cases selected from a test suite
Fitness Measure	Weighted sums bi-criteria model (Section 4.2.2)
Heuristic Operators	Elitist Selection
	Roulette Wheel Selection
	Mutation
	Single-point Crossover
	Growth (Addition and Deletion of Chromosomes)
Control Operators	Experiment values listed in Table II
Termination Criteria	Completion of g_{max} generations, $g_{max} \in \{25, 50, 75\}$

Table 4.1: GA Problem Formulation and Configuration.

dependencies. This requirement enables the prioritizer to reorder tests into any sequence that maximizes the suite's ability to isolate defects.

4.2.1 Overview

A genetic algorithm is used to heuristically solve Problem 1. First, the execution time of each test case is recorded. Because a time constraint could be very short, test case execution times must be exact in order to properly perform a selection. Timing information additionally includes any initialization and shutdown time required by a test.

A program P and each $T_i \in \langle T_1, \ldots, T_n \rangle$ are input into the genetic algorithm, along with the following user specified parameters: (i) s, size of the population, (ii) g_{max}, maximum number of generations, (iii) p_t, percent of the execution time of T allowed by the time budget, (iv) p_c, crossover probability, (v) p_m, mutation probability, (vi) p_a, addition probability, (vii) p_d, deletion probability, (viii) tc, test adequacy criterion, and (ix) w, program coverage weight, where $p_t, p_c, p_m, p_a, p_d \in [0, 1]$. The genetic algorithm uses heuristic search to solve Problem 1 and to identify the test tuple $\sigma_{max} \in perms(2^T)$ that is likely to have the fastest rate of fault detection in the provided time limit. In general, any $\sigma_j \in perms(2^T)$ has the form $\sigma_j = \langle T_i, \ldots, T_u \rangle$ where $u \leq n$. Table 4.1 describes our genetic algorithm's problem formulation.

4.2.2 A Genetic Algorithm for Time-Aware Test Selection

We design our GASelect algorithm as shown in Figure 4.2. GASelect prioritizes test suite T based on a given time constraint p_t, as described in Problem 1. On line 1, this

Algorithm GASELECT($P, T, s, g_{max}, p_t, p_c, p_m, p_a, p_d, tc$)

Input: Program P, Test suite T, Population size s, Maximum generations g_{max}, Percent of test suite time p_t, Crossover probability p_c, Mutation probability p_m, Addition probability p_a, Deletion probability p_d, Test adequacy criteria tc,

Output: Maximum fitness tuple $F_{max} \in F$ in set σ_{max}

1. $t_{max} \leftarrow p_t \times \sum_{i=0}^{n} time(\langle T_i \rangle)$
2. $R_0 \leftarrow \emptyset$
3. **repeat**
4. $\quad R_0 \leftarrow R_0 \cup \{CreateRandomIndividual(T, p_t)\}$
5. **until** $|R_0| = s$
6. $g \leftarrow 0$;
7. **repeat**
8. $\quad F \leftarrow \emptyset$
9. $\quad$ **for** $\sigma_j \in R_g$
10. $\quad\quad F \leftarrow F \cup \{CalcFitness(P, \sigma_j, t_{max}, tc)\}$
11. $\quad R_{g+1} \leftarrow ElitistSelect(R_g, F)$
12. $\quad$ **repeat**
13. $\quad\quad \sigma_k, \sigma_l \leftarrow RouletteWheelSelect(R_g, F)$
14. $\quad\quad \sigma_q, \sigma_r \leftarrow ApplyCrossover(p_c, \sigma_k, \sigma_l)$
15. $\quad\quad \sigma_q \leftarrow ApplyMutation(p_m, \sigma_q)$
16. $\quad\quad \sigma_r \leftarrow ApplyMutation(p_m, \sigma_r)$
17. $\quad\quad \sigma_q \leftarrow AddAdditionalTests(T, p_a, \sigma_q)$
18. $\quad\quad \sigma_r \leftarrow AddAdditionalTests(T, p_a, \sigma_r)$
19. $\quad\quad \sigma_q \leftarrow DeleteATest(p_d, \sigma_q)$
20. $\quad\quad \sigma_r \leftarrow DeleteATest(p_d, \sigma_r)$
21. $\quad\quad R_{g+1} \leftarrow R_{g+1} \cup \{\sigma_q\} \cup \{\sigma_r\}$
22. $\quad$ **until** $|R_{g+1}| = s$
23. $\quad g \leftarrow g + 1$
24. **until** $g > g_{max}$
25. $\sigma_{max} \leftarrow FindMaxFitnessTuple(R_{g-1}, F)$
26. **return** σ_{max}

Figure 4.2: The GA Selection Algorithm.

algorithm calculates p_t, percent of the total time of T, and stores the value in t_{max}, the maximum execution time for a tuple. In the loop beginning on line 3, the algorithm creates a generation R_0 containing s random test tuples σ from $perms(2^T)$ that can be executed in t_{max} time. Each set of s individuals make up a population.

R_0 is the first generation of s potential solutions to Problem 1. Once a generation of test tuples is created, coverage information is used by the $CalcFitness(P, \sigma_j, t_{max}, tc)$ method on line 10. The $CalcFitness(P, \sigma_j, t_{max}, tc)$ method is used to determine the "goodness" of σ_j. To simplify the notation, we denote F_j the fitness value of σ_j, where $F_j = CalcFitness(P, \sigma_j, t_{max}, tc)$. We also use $F = \langle F_1, F_2, \ldots, F_s \rangle$ to denote the tuple of fitnesses for each $\sigma_j \in R_g, 0 \leq g \leq g_{max}$.

The $ElitistSelect(R_g, F)$ method on line 11 chooses the two best test tuples in R_g to be elements in the next generation R_{g+1} of test tuples. The two best tuples are chosen in order to guarantee that R_{g+1} has at least one "good" pair. It is important to carry these highly fit tuples into R_{g+1} as they are in R_g because they are most likely very close to exceeding t_{max}. Any slight change to these test tuples could cause them to require too much execution time, thus invalidating them. Since the GA is trying to identify one particular test tuple, this elitist selection technique ensures that the best tuple in R_g survives on to R_{g+1} [37].

On line 13, $RouletteWheelSelect(R_g, F)$ identifies pairs of tuples $\{\sigma_k, \sigma_l\}$ from R_g through a roulette wheel selection technique based on a probability proportional to $|F|$. This implies that candidate solutions with higher fitnesses are more likely to be selected than those with low fitnesses. In this technique, the fitness values are first normalized in relation to the rest of the test tuple set so that the sum of all fitness values equals one [37]. The test tuples are then sorted by descending fitness values, and accumulated normalized fitness (ANF) values are calculated. A random number $r \in [0, 1]$ is next generated, and the first individual whose accumulated normalized value is greater than or equal to r is selected. This selection method is repeated until enough tuples are selected to fill the generation R_{g+1}. Candidate individuals with higher fitnesses are therefore less likely to be eliminated, but a few with lower fitness have a chance to be used to make the next population as well [37].

The $ApplyCrossover(p_c, \sigma_k, \sigma_l)$ method on line 14 may merge the pair $\{\sigma_k, \sigma_l\}$ to create two potentially new tuples $\{\sigma_q, \sigma_r\}$ based on p_c, a user given crossover probability. Each tuple in the pair $\{\sigma_q, \sigma_r\}$ may then be mutated based on p_m, a user provided mutation probability. Test cases are then added or deleted from σ_q or σ_r using the $AddAdditionalTests(T, p_a, \sigma_r)$ and $DeleteATest(p_d, \sigma_r)$ methods. The crossover operator exchanges subsequences of the test tuples, and the mutation operator only mutates single elements. Test case addition and deletion are needed because no other operator allows for a change in the number of test cases in a tuple.

After GASelect makes each of these modifications to the original pair, both tuples σ_q and σ_r are entered into R_{g+1}, as seen on line 21. The algorithm applies the same transformations to all pairs selected by the $RouletteWheelSelection(R_g, F)$ method until R_{g+1} contains s test tuples. In total, g_{max} populations of s test tuples are iteratively created in this fashion as specified in Figure 4.2 on lines 7–24. After GASelect creates the final generation of $R_{g_{max}}$, line 25 identifies σ_{max}, the test tuple with the greatest fitness. This tuple is guaranteed to be the tuple with the highest fitness out of all g_{max} populations due to the elitist strategy.

Test Coverage. Since it is very rare for a tester to know the location of all faults in P prior to testing, the selection technique must estimate how likely a test is to find defects, which factors into the function fit of Problem 1. Recall that the function fit yields the fitness of the tuple σ_j based on its potential for fault detection and its time consumption. As it is impossible to reveal a fault without executing the faulty code, the percent of code covered by a test suite is used to estimate the suite's potential. In this chapter, two forms of test adequacy criteria tc are considered: (i) method coverage and (ii) block coverage [27, 49, 91].

Our genetic algorithm accepts coverage information based on the aggregate coverage of a test suite rather than per-test data. As noted by Kessis et al., this is the form that many tools such as Clover [53], Jazz [76], and Emma [91] produce. Thus, our selection approach can reorder a test suite without requiring per-test coverage information. While the genetic

algorithm handles the common case, its calculation of test tuple fitness could be enhanced to use coverage information on a per-test basis. This would dramatically improve the time overhead of the fitness function, as explained in Section 4.3.2.

Fitness Function. The $CalcFitness(P, \sigma_j, t_{max}, tc)$ method on line 10 uses t_{max} and $fit(P, \sigma_j, tc)$ to calculate fitness. The fitness function, represented by fit in Problem 1, assigns each test tuple a fitness based on (i) the percentage of requirements covered in P by that tuple and (ii) the time at which each test covers its associated code in P.

This function equates to the order-aware coverage metric described and used in Chapter 3.3.2. The primary fitness $F_{pri} \in [0, 100]$ is calculated by measuring the code coverage cc of the test selection σ_j. The second component $F_{sec} \in [0, 1]$ considers the incremental code coverage of the tuple, giving preference to test selections whose earlier tests have greater coverage. The secondary fitness is calculated as:

$$\begin{aligned} F_{s-actual}(P, \sigma_j, tc) &= \sum_{i=1}^{|\sigma_j|} time(\langle T_i \rangle) \times cc(P, \sigma_{j\{1,i\}}, tc) \\ F_{s-max}(P, \sigma_j, tc) &= cc(P, \sigma_j, tc) \times \sum_{i=1}^{|\sigma_j|} time(\langle T_i \rangle) \\ F_{sec}(P, \sigma_j, tc) &= \frac{F_{s-actual}(P, \sigma_j, tc)}{F_{s-max}(P, \sigma_j, tc)} \end{aligned}$$

for $\sigma_j \in perms(2^T)$.

Then

$$fit(P, \sigma_j, tc) = F_{pri}(P, \sigma_j, tc) + F_{sec}(P, \sigma_j, tc) \tag{4.1}$$

If a test tuple execution time $time(\sigma_j)$ is greater than the time budget t_{max}, F_j is automatically set to -1 by the $CalcFitness(P, \sigma_j, t_{max}, tc, w)$ method. Because such a tuple violates the execution time constraint, it cannot be a solution and thus receives the worst fitness possible. While a tuple σ_j with $F_j = -1$ could simply not be added to the next generation R_{g+1}, populations with individuals that have a fitness of -1 can actually be favorable. Since the "optimal" test tuple selection likely teeters on the edge of exceeding

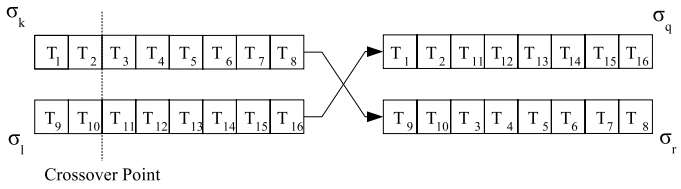


Figure 4.3: Crossover with Random Crossover Point.

the designated time budget, any slight change to a σ_j with $F_j = -1$ could create a new valid test tuple. Therefore, σ_j's with $F_j = -1$ are maintained for possible selection just like any other generated selection. If the test tuple execution time $time(\sigma_j) \leq t_{max}$, Equation 4.1, $fit(P, \sigma_j, tc)$ is used to calculate fitness.

Crossover. The GA uses crossover to vary test tuples from one test tuple generation to the next through recombination. Pairs of test tuples $\{\sigma_k, \sigma_l\}$ are selected out of R_g. The $ApplyCrossover(p_c, \sigma_k, \sigma_l)$ method performs crossover to create two potentially new hybrid test tuples from $\{\sigma_k, \sigma_l\}$. First, a random number $r_1 \in [0, 1]$ is generated. If r_1 is less than the user provided value for p_c, the crossover operator is applied. Otherwise, the parent individuals are unchanged and await the next step, mutation. If crossover is to occur, the $ApplyCrossover(p_c, \sigma_k, \sigma_l)$ method on line 14 selects another random number $r_2 \in [0, min(|\sigma_k|, |\sigma_l|)]$ as the crossover point, where $|\sigma_k|$ and $|\sigma_l|$ are the number of test cases in σ_k and σ_l, respectively. The subsequences before and after the crossover point are then exchanged to produce two new offspring, as seen in Figure 4.3. If crossover causes two of the same test cases to be in the same test tuple, another random test not in the current tuple is selected from T instead of including the duplicated test case.

Mutation. The use of the $ApplyMutation(p_m, \sigma_j)$ method on lines 15 and 16 of Figure 4.2 also provides a way to add variation to a new population. The new test tuple is identical to the prior parent tuple except that one or more changes may be made to the new tuple's test cases. All test tuples that are selected on line 13 are first considered for crossover. Then they are subject to mutation at each test case position with a small user specified mutation probability p_m. If a random number $r_3 \in [0, 1]$ is generated such that r_3 is less than p_m for test case T_i, a new test not included in the current test tuple is randomly

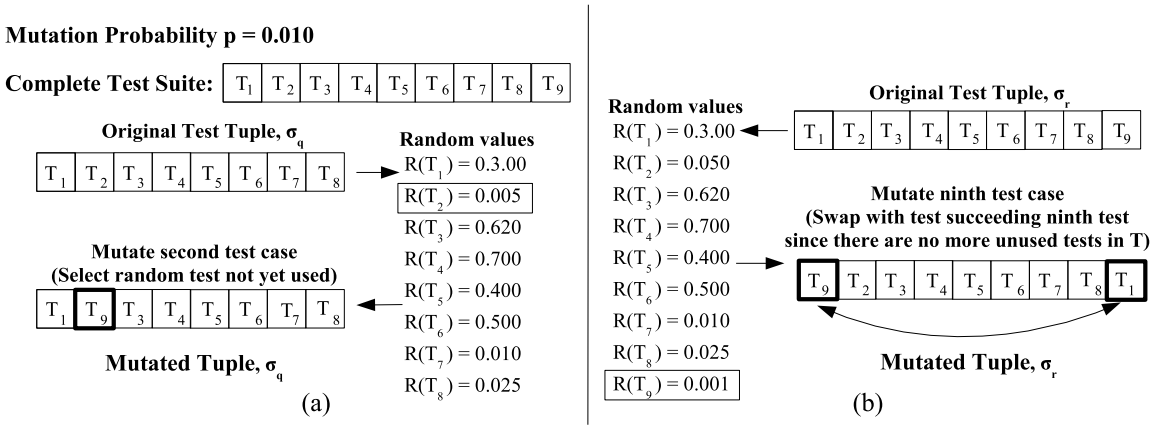


Figure 4.4: Mutation of a Test Tuple.

selected from T to replace T_i, as demonstrated for T_2 in Figure 4.4(a). Figure 4.4(b) also shows that if there are no unused tests in T when T_9 is chosen for mutation, the test tuple is still mutated. Instead of replacing the test with a random test, the test is swapped with the test case that succeeds it.

Addition and Deletion Test cases can also be added to or deleted from the test tuples using the $AddAdditionalTests(T, p_a, \sigma_j)$ method on lines 17 and 18 or the $DeleteATest(p_d, \sigma_j)$ method on lines 19 and 20. As in messy genetic algorithms [38], the sets of tuples R_g must be allowed to grow beyond the initial generation R_0. Addition and deletion features permit such growth. While the crossover operator exchanges subsequences, it does not increase the number of test cases within an individual. Similarly, the mutation operator only mutates single elements at each index within the test tuple. Although addition and deletion operations are necessary, they should be performed infrequently so as to not violate the genetic algorithm's selection, crossover, and mutation techniques. If a random number $r_4 \in [0, 1]$ is generated such that $r_4 < p_a$, a random test case is removed from the individual. If another random number $r_5 \in [0, 1]$ is generated and $r_5 < p_d$, a random test case not yet executed in the individual is added to the end of the test sequence.

4.2.3 Test Selection in Action

To demonstrate the evolution of individuals in a population over generations by means of selection, mutation, crossover, and growth functions, consider the following example execution. Two generations are described with five individuals in each population. The crossover probability is $p_c = 0.7$, the mutation probability is $p_m = 0.1$, and the addition/deletion probabilities are $p_a = p_d = 0.02$. The `Gradebook` application, which was described in Chapter 3.3.1, contains 28 test cases. We call these $T1$ through $T28$ for simplicity. The time constraint is set to 7008 ms or 50% of the execution time of these 28 tests.

As described in Section 4.2.2, the algorithm creates the first individual by randomly adding test cases into the test tuple until the tuple exceeds the time constraint. This is performed five times, once for each individual. The example begins with the following initial population:

$R_0[0] = \langle T26, T22, T19, T28, T11, T2, T16, T15, T23, T20 \rangle$	Fitness:	49.20
$R_0[1] = \langle T5, T2, T22, T8, T27, T6, T9, T18, T28, T3, T17, T14, T20, T15, T21 \rangle$	Fitness:	60.19
$R_0[2] = \langle T6, T2, T4, T9, T25, T11, T15, T5, T8, T21, T1, T28, T27, T19, T23, T13 \rangle$	Fitness:	58.58
$R_0[3] = \langle T11, T3, T4, T1, T8, T21, T12, T18, T6, T20, T23, T5, T19, T14, T13, T27, T9 \rangle$	Fitness:	50.64
$R_0[4] = \langle T16, T23 \rangle$	Fitness:	19.91

To begin generating the next population, two individuals are selected using an elitist strategy. The second generation then has two of the five necessary individuals.

$R_1[0] = \langle T5, T2, T22, T8, T27, T6, T9, T18, T28, T3, T17, T14, T20, T15, T21 \rangle$	Fitness:	60.19
$R_1[1] = \langle T6, T2, T4, T9, T25, T11, T15, T5, T8, T21, T1, T28, T27, T19, T23, T13 \rangle$	Fitness:	58.58

Two individuals are then chosen using roulette wheel selection [37]. For roulette wheel selection, the fitness values are first normalized. In this example, the fitness values are normalized by multiplying by $1/(49.20 + 60.19 + 58.58 + 50.64 + 19.91) = 0.00419$. Then the individuals are sorted by fitness, and the ANF values are calculated.

$R_0[4] = \langle T16, T23 \rangle$	ANF:	0.0834
$R_0[0] = \langle T26, T22, T19, T28, T11, T2, T16, T15, T23, T20 \rangle$	ANF:	0.2897
$R_0[3] = \langle T11, T3, T4, T1, T8, T21, T12, T18, T6, T20, T23, T5, T19, T14, T13, T27, T9 \rangle$	ANF:	0.5020
$R_0[2] = \langle T6, T2, T4, T9, T25, T11, T15, T5, T8, T21, T1, T28, T27, T19, T23, T13 \rangle$	ANF:	0.7476
$R_0[1] = \langle T5, T2, T22, T8, T27, T6, T9, T18, T28, T3, T17, T14, T20, T15, T21 \rangle$	ANF:	1.0

The random number 0.802 is generated, selecting $R_0[1]$ as σ_k. $R_0[3]$ is also chosen as σ_l from a second random number draw of 0.496. The genetic algorithm operators including crossover, mutation, addition, and deletion may now be performed based on random numbers. First, a random number, $r_1 = 0.6$, is chosen for crossover. Since $0.6 < p_c = 0.7$, crossover is performed. Because there are 15 test cases in the smaller test tuple σ_k, a random number $r_2 \in (0, 15) = 5$ is selected as the crosspoint, which creates the following two new individuals.

$$\sigma_q = \langle T11, T3, T4, T1, T8, T21, T9, T18, T28, T3, T17, T14, T20, T15, T21 \rangle$$
$$\sigma_r = \langle T5, T2, T22, T8, T27, T6, T12, T18, T6, T20, T23, T5, T19, T14, T13, T27, T9 \rangle$$

However, these individuals contain repeating test cases. For example, σ_q contains the test T3 twice. As the test cases are assumed to be independent, there is no reason to run any test case more than once. Thus, the repeating test cases are replaced with random unused test cases from T, as seen below in bold.

$$\sigma_q = \langle T11, T3, T4, T1, T8, T21, T9, T18, T28, \mathbf{T5}, T17, T14, T20, T15, \mathbf{T24} \rangle$$
$$\sigma_r = \langle T5, T2, T22, T8, T27, T6, T12, T18, \mathbf{T10}, T20, T23, \mathbf{T7}, T19, T14, T13, \mathbf{T21}, T9 \rangle$$

These newly created individuals are next mutated. A random number $r_{3_i} \in [0, 1]$ is generated for each test case T_i in σ_q and σ_r. If r_{3_i} is less than $p_m = 0.1$, a new test not included in the current test tuple is randomly selected from T to replace T_i. For example, in σ_q, $r_{3_0} = 0.82$, $r_{3_1} = 0.13$, and $r_{3_2} = 0.04$. Thus, $T_0 = T11$ and $T_1 = T3$ remain unchanged, while $T_2 = T4$ is mutated to $T6$. After drawing random numbers for each test in σ_q and σ_r, the mutation method calls on lines 15 and 16 give the following new mutated individuals:

$$\sigma_q = \langle T11, T3, T4, T1, T8, T21, T9, T18, T28, T5, T17, T14, T20, T15, T24 \rangle$$
$$\rightarrow \langle T11, T3, \mathbf{T6}, T1, T8, T21, T9, T18, \mathbf{T2}, T5, T17, T14, T20, T15, T24 \rangle$$
$$\sigma_r = \langle T5, T2, T22, T8, T27, T6, T12, T18, T10, T20, T23, T7, T19, T14, T13, T21, T9 \rangle$$
$$\rightarrow \text{(no change)}$$

Finally, a random test may be added or removed from the new individuals through the selection of two more random numbers r_4 and r_5. When $r_4 < p_a$, a random unused test case is picked from T and placed at the end of the individual, as observed in σ_q where $r_4 = 0.014$. For σ_r, $r_4 \geq p_a$, but r_5 was less than p_d, causing a random test, $T5$, to be removed from tuple σ_r.

$$\sigma_q = \langle T11, T3, T4, T1, T8, T21, T9, T18, T28, T5, T17, T14, T20, T15, T24, \mathbf{T2} \rangle$$
$$\sigma_r = \langle T2, T22, T8, T27, T6, T12, T18, T10, T20, T23, T7, T19, T14, T13, T21, T9 \rangle$$

The two individuals σ_q and σ_r are now finally added to the second population as the third and fourth test tuples, $R_1[2]$ and $R_1[3]$. This process is repeated to obtain the fifth individual, $R_1[4] = \langle T23, T16 \rangle$, which is simply the reverse of the original fifth individual $R_0[4]$. However, the fitness of the new individual $R_1[4]$ is higher because it covers more requirements earlier than $R_0[4]$. In this way, the second generation is completed and contains the following individuals:

$R_1[0] = \langle T5, T2, T22, T8, T27, T6, T9, T18, T28, T3, T17, T14, T20, T15, T21 \rangle$	Fitness:	60.19
$R_1[1] = \langle T6, T2, T4, T9, T25, T11, T15, T5, T8, T21, T1, T28, T27, T19, T23, T13 \rangle$	Fitness:	58.58
$R_1[2] = \langle T11, T3, T4, T1, T8, T21, T9, T18, T28, T5, T17, T14, T20, T15, T24, T2 \rangle$	Fitness:	68.13
$R_1[3] = \langle T2, T22, T8, T27, T6, T12, T18, T10, T20, T23, T7, T19, T14, T13, T21, T9 \rangle$	Fitness:	50.90
$R_1[4] = \langle T23, T16 \rangle$	Fitness:	20.02

In the first population, the average fitness was 47.704, while that of the second population was 51.564. The genetic algorithm operators produced new individuals with fitnesses higher than those of the original population in this example. Note that this may not always be the case. However, it is guaranteed that the individuals with the highest fitnesses will survive from generation to generation, continuing to contribute good genes to the population.

4.3 Empirical Evaluation

The primary goal of our experimental study is to identify and evaluate the challenges associated with test suite selection given a time budget. We implemented the approach described in Section 4.2 in order to measure its effectiveness and efficiency. The goals of the experiment are as follows:

1. Analyze trends in the (i) coverage preservation, (ii) overall coverage, (iii) fitness, and (iv) average percent of faults detected by selections generated using different values for the parameters of the genetic algorithm.

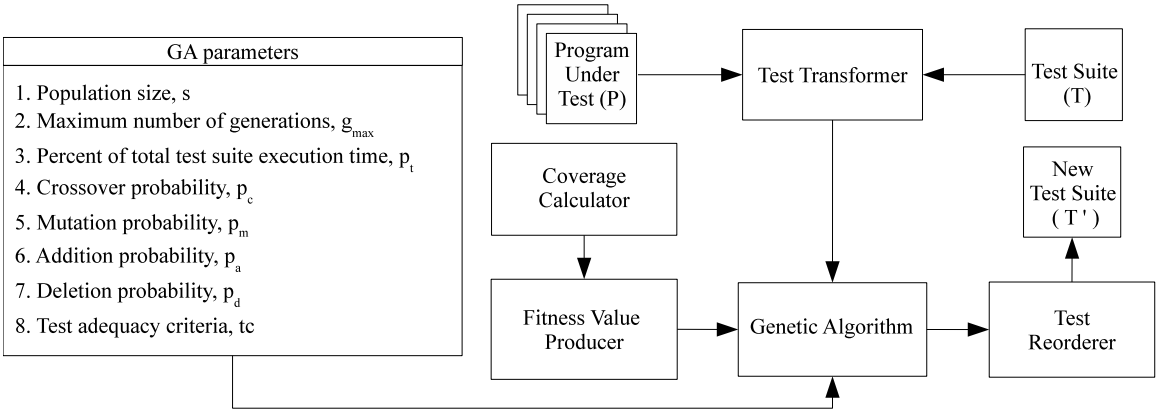


Figure 4.5: Overview of Selection Infrastructure.

2. Determine if the GA-produced selections, on average, outperformed selections produced by other prioritization techniques according to the average percent of faults detected.

3. Identify the trade-offs between the configuration of the genetic algorithm and the time and space overheads associated with the creation of the selected test suite.

4.3.1 Experimental Design

All experiments were performed on GNU/Linux workstations with kernel 2.4.20-8, a 1.80 GHz Intel Pentium 4 processor and 1 GB of main memory. The genetic algorithm was implemented in Java and selects JUnit test suites, which are commonly used in testing [27]. Figure 4.5 provides an overview of the test selection implementation with edges between interacting components. The test suite is first transformed into a set of test cases with test case execution times. JUnit's test execution framework provides `setUp` and `tearDown` methods that are used to set and clear application state, transforming Δ_{i-1} into Δ_0. The `tearDown` operation is also used to store application state Δ_i prior to deletion. Thus, Section 4.2's assumption of test independence is acceptable. To begin GA execution, the test cases and program information are input into the genetic algorithm along with the other nine parameters for the GA, as depicted in Figure 4.5.

Faults	Test Cases						
	T_1	T_2	T_3	T_4	T_5	T_6	T_7
ϕ_1	X			X			
ϕ_2			X				
ϕ_3		X					X
ϕ_4				X			
ϕ_5		X				X	

Table 4.2: Faults Detected by $T = \langle T_1, \ldots, T_7 \rangle$.

The genetic algorithm gathers coverage information at most $|\sigma_j|$ times whenever the fitness of test tuple σ_j is calculated. Fitness is determined before any test tuple is added to the next test tuple set R_g. Emma, an open source toolkit for reporting Java code coverage, is used to calculate test adequacy in terms of method and block coverage [91]. Coverage statistics are aggregated at method, class, package, and all classes levels for the application under test, and Emma, like most tools, only reports coverage for the entire test tuple. The overall runtime overhead of instrumentation added by Emma is small and Emma's bytecode instrumentor itself is very fast, mostly limited by file input/output (I/O) speed [91].

Coverage calculation is expensive due to the number of times coverage information must be gathered. In order to prevent redundant coverage calculations, the infrastructure uses memoization [72]. This is especially useful in the calculation of the secondary fitness function F_{sec}, which requires the code coverage information for up to $|\sigma_j|$ subtuples of test cases for each $\sigma_j \in R_g$. Coverage information is used in the fitness function to calculate a fitness value $fit(P, \sigma_j, tc)$ for every $\sigma_j \in R_g$. Based on this value, the GA creates g_{max} sets of s test tuples. From the last generated test tuple set, the test tuple with the maximum fitness σ_{max} is returned. As seen in Figure 4.5, σ_{max} is then used in the new test suite T'.

`Gradebook` and `JDepend`, described in Chapter 3.3.1, are again used as case study applications.

Evaluation Metrics.

In order to evaluate the effectiveness of a given tuple of test cases, the overall code coverage and fitness of the GA's resulting test tuples is used. Additionally, the amount of coverage preserved by each of the GA's resulting test tuples from the original test suite is determined. The coverage information is analyzed at block granularity. For a more thorough analysis of the effectiveness of a given test case tuple, prior knowledge of the faults within the program under test is assumed. A test suite selection can be empirically evaluated based on the weighted average of the percentage of faults detected over the execution of the test suite, or the *APFD*, described in Chapter 2.1.2. Preference is given to selection schemes that produce test suite tuples with high APFD values.

Since σ_j is a subtuple of T, it may contain fewer test cases than T. Moreover, σ_j may not be able to detect all defects. Therefore, we extend the definition of *reveal* so that $reveal(\phi_f, \sigma_j) = |\sigma_j| + 1$ if a fault ϕ_f was not found by any test case in σ_j. This would cause a prioritized test suite tuple that finds few faults to possibly have a negative APFD. Suites finding few faults are penalized in this way.

For example, suppose that we have the test suite $T = \langle T_1, \ldots, T_7 \rangle$ for program P, and we know that the tests detect faults $\Phi = \{\phi_1, \ldots, \phi_5\}$ in P according to Table 4.2. We look at two scenarios: In the first, P contains a subset of the Φ faults, namely $\Phi_1 = \{\phi_2, \phi_3\}$. In the second, P is seeded with all of the Φ faults, so $\Phi_2 = \Phi$. Consider the two prioritized test tuples $\sigma_1 = \langle T_3, T_2, T_1, T_6, T_4 \rangle$ and $\sigma_2 = \langle T_1, T_5, T_2, T_4 \rangle$. Incorporating the data from Table 4.2 into the APFD equation yields

$$\text{APFD } (\sigma_1, P, \Phi_1) = 1 - \tfrac{1+2}{5\times 2} + \tfrac{1}{2\times 5} = 0.80$$
$$\text{APFD } (\sigma_2, P, \Phi_1) = 1 - \tfrac{5+3}{4\times 2} + \tfrac{1}{2\times 4} = 0.125$$
$$\text{APFD } (\sigma_1, P, \Phi_2) = 1 - \tfrac{3+1+2+5+2}{5\times 5} + \tfrac{1}{2\times 5} = 0.58$$
$$\text{APFD } (\sigma_2, P, \Phi_2) = 1 - \tfrac{1+5+3+4+3}{4\times 5} + \tfrac{1}{2\times 4} = 0.325$$

Note that σ_2 is penalized because it fails to find ϕ_2. Also, APFD varies depending on the set of faults that are seeded into P. According to the APFD metric, σ_1 has a better percentage

of fault detection than σ_2 in both scenarios and is therefore more desirable for both fault sets.

To evaluate the efficiency of our approach, time and space overheads are analyzed by using a GNU/Linux process tracking tool. This tool supports the calculation of peak memory use and the total user and system time required to prioritize the test suite.

4.3.2 Experiments and Results

Experiments were run in order to analyze (i) the effectiveness and the efficiency of the parameterized genetic algorithm and (ii) the effectiveness of the genetic algorithm in relation to random, initial ordering, reverse ordering, and fault-aware selections.

Fault Seeding. In order to calculate APFD , the resulting test selections were run on programs that were seeded with sets of faults created by a tool, Jester [78]. Mutations were determined by a mutation configuration file, which contains value substitutions such as replacing '`+`' by '`-`' or '`>`' by '`<`'. The default mutations provided by Jester were used [78]. For each mutation and each source file in program P, Jester modifies the file, recompiles P, and executes the test suite T. P is then returned to its original state for the next seeding. For the purpose of the experiments, Jester was modified to show all of the mutations that were successfully found by a test T_i in T. In this way, Jester generated a list of faults that are detectable by at least one test case in the test suites of `JDepend` and `Gradebook`.

For example, in the `Gradebook` application, the following mutations were suggested by Jester and detectable by `Gradebook`'s test suite:

GradeBookCreator:25	`databaseServerCreated = false;`	`databaseServerCreated = true;`
GradeBookCreator:407	`"Firsthand Archer(255), "`	`"Firsthand Archer(355), "`
Gradebook:1001	`if( studentId < 0 )`	`if(true || studentId < 0 )`

Similar mutations, like the ones below, were made in `JDepend`:

jdepend.framework.FileManager.java:74	`if (acceptFile(file)) {`	`if !(acceptFile(file)) {`
jdepend.framework.JavaPackage:113	`count++;`	`count--;`

GA parameters	
P	Gradebook, JDepend
$(g_{max},\ s)$	$(25, 60),\ (50, 30),\ (75, 15)$
p_t	$0.25, 0.50, 0.75$
p_c	0.7
p_m	0.1
p_a	0.02
p_d	0.02
tc	method, block

Table 4.3: Parameters used in GA Configurations.

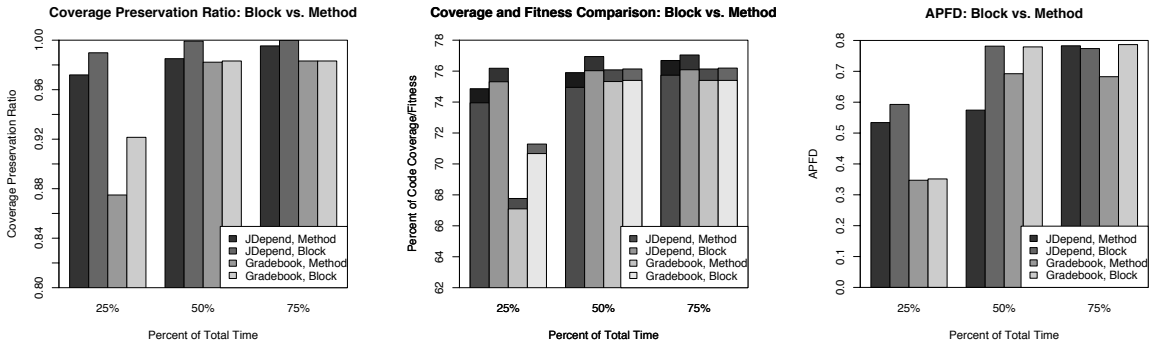
For each application, 40 mutations that could be found by at least one $T_i \in \langle T_1, \ldots, T_n \rangle$ were randomly selected. We could look at these 40 faults in a table like that in Table 2.1. Then, to calculate APFD, 25, 50, or 75% of the 40 possible mutations were seeded into the case study applications, where the larger mutation sets were supersets of the smaller mutation sets.

The first experiment compares the GA execution results and overheads from different GA parameter configurations, described in Table 4.3. These parameters were chosen based on past work [77] and were shown to be good in a preliminary study [114]. In order to run all possible configurations, 36 experiments were completed: 18 using `Gradebook` and 18 using `JDepend`. We used thirty-six computers, each running one trial with one unique configuration. For example, one computer ran a genetic algorithm on the test suite T of the `Gradebook` application calculating $g_{max} = 25$ generations of tuple sets, each of which contained $s = 60$ test tuples. In this configuration, the selection was created to be run with $p_t = 0.25$, requiring solution test tuples to execute within 25% of the total execution time of T, and fitness was measured using method coverage. The first set of experiments evaluated the effectiveness of the selections produced by the GA. We also analyzed the efficiency of the technique, making this study one of the first to empirically evaluate the efficiency of a search-based testing technique and thus provide concrete evidence of the intuitions developed by Harman in [40].

Effectiveness. As shown in Table 4.4, on average, the selections created with fitnesses

	Block	Method
`Gradebook`	0.638993	0.573982
`JDepend`	0.715984	0.630298

Table 4.4: `Gradebook` and `JDepend` APFD Values.



(a) (b) (c)

Figure 4.6: GA Coverage Preservation, Coverage/Fitness, and APFD Results.

based on block coverage outperformed those developed with fitnesses based on method coverage. In `Gradebook`, use of block coverage produced APFD values 11.32% greater than the use of method coverage, and in `JDepend`, block coverage APFD values increased by 13.59% over method coverage. We attribute this to block coverage's finer level of granularity.

The same trend is observed when looking at the code coverage and fitness of each of the test selections. As shown in Figure 4.6(a), selections generated with fitnesses based on block coverage preserved a larger percent of the original test suite's coverage. However, each selection maintained code coverage well, with the worst test tuple still achieving 87.5% of the original code coverage. Even when the allotted time is reduced by 75%, our technique on average preserves 94% of the original test suite's code coverage. Preservation of 100% is not possible for `Gradebook` because no possible selection that covers all of T's covered requirements can be run within any of the time budgets. `Gradebook`'s test cases in general are longer running than those of `JDepend`.

Figure 4.6(b) shows the code coverage and fitness of each of the selections, where the lower bar is the overall coverage of the test tuple and the higher bar is the fitness. Fig-

ure 4.6(c) provides the APFD of each tuple. As the time budget is increased, the coverage, fitness, and APFD values increase for both `Gradebook` and `JDepend`, although the amount of increase for a `JDepend` selection is less than that of the `Gradebook` selections. The `Gradebook` test cases that cover the most code and that find the most faults take a significantly longer time to execute than the test cases of `JDepend`. Thus, fewer `Gradebook` test cases that cover larger portions of the code can be executed within a shorter time budget of 25%, as observed in Figures 4.6(a) and 4.6(b). This causes the APFD of the `Gradebook` selections with $p_t = 0.25$ to be lower than the rest of the resulting test tuples. When p_t is increased to 50%, the majority of the test cases that find the most faults are able to be run within the time budget, which greatly increases test tuple APFD values. An increase to $p_t = 0.75$ allows for the inclusion of the shorter, less useful test cases.

`JDepend`'s test cases all have very short execution times, and many of them cover about the same amount of code. As in `Gradebook`, the longer running `JDepend` test cases generally cover more code and detect more faults than the shorter tests. However, because the execution time difference between `JDepend` test cases is much smaller than that of `Gradebook` test cases, we observe a less drastic coverage and APFD increase in `JDepend`'s selections as p_t grows. This can be seen in Figures 4.6(b) and 4.6(c), especially between $p_t = .25$ and $p_t = .50$.

Modification of the number of faults seeded and of (g_{max}, s) led to APFD values that were nearly constant in terms of block and method coverage in the test selections for `Gradebook` and `JDepend`. This provides confidence in the results generated by the GA because about the same percentage of defects can be found by any of the selections regardless of how many faults there are or how the GA created the selections. Just as in Table 4.4 and Figure 4.6(c), selections based on block coverage slightly outperformed those using method coverage.

Efficiency. Space costs were insignificant, with the peak memory use of any experiment being less than 9344 KB. Most experiments ran with peak memory use of approximately 1344 KB. As is seen in Figure 4.7, the number of generations and the number of tuples per generation greatly impact the time overhead. Modifying these parameters did not affect

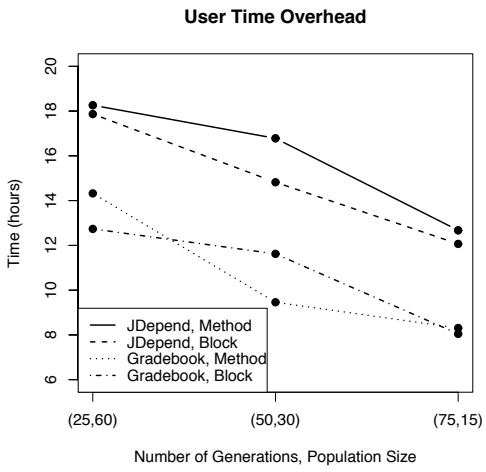


Figure 4.7: Genetic Algorithm Time Results.

the quality of the solutions [114]. The high time overheads provide quantitative evidence of the prevailing understanding of the high cost of search-based methods [40]. For example, using block coverage, the genetic algorithm's selection of `Gradebook`'s test suite executed for 12.7 hours of user time on average when creating 25 generations of 60 test tuples. On the other hand, if 75 generations with 15 test tuples were created, the GA only consumed 8.0 hours. Due to memoization, many of the fitness values of test subtuples created in later GA iterations were already recorded from earlier iterations so that the fitness of the subtuples did not need to be calculated again. In the experiments that created 25 generations of 60 test tuples, there is likely to be more genetic diversity. Thus, there are more subsequences that will probably be found than when selection is performed with 75 generations of 15 test tuples. In this case, Emma must be run many more times, which increases the selection time overhead.

The same trend observed in Figure 4.7 occurs when the system time values for `Gradebook` and `JDepend` are compared [114]. For example, a GA executing `Gradebook`'s test suite with 25 generations of 60 test tuples using block coverage required 12.7 hours of user time and 0.78 hours of system time. However, a GA running `Gradebook`'s test suite with 75 generations of 15 individuals required only 8.0 hours of user time and 0.44 hours of system time, a

vast improvement over the $(25, 60)$ configuration. Time-aware selection of the `JDepend` test suite consumed 17.9 hours of user time and 2.1 hours of system time when using the $(25, 60)$ configuration but only 12.1 hours of user time and 1.38 hours of system time using $(75, 15)$. A GA prioritizing `JDepend`'s test suite requires a longer execution time than a GA prioritizing the `Gradebook` test suite due to `JDepend`'s larger test suite. Since there are more selections that can be generated, on average, the fitness function was calculated more times.

As the percent of total test suite execution time was increased for both `Gradebook` and `JDepend`, the number of fitness function calculations also grew due to the fact that more test cases could be included in the selections. Since profiling reveals that the fitness function is the main bottleneck of the technique, less time is required overall to reach a result when the genetic algorithm needs to run the fitness function calculator less frequently. This confirms the trend seen in Figure 4.7 as well. We also note that no significant difference was observed between the time overheads of test suite selection using block versus method coverage.

The time overhead of the genetic algorithm is dominated by the execution of Emma, which executes subsets of T. This is done because we assume that only aggregate coverage information is available, as discussed in Section 4.2.2. Even when performing memoization, Emma could be executed $O\left(\sum_1^{n-1} C(n,i)\,(n-i)!\right) = O\,(n!)$ times, where $n = |\,\langle T_1, \ldots, T_n\rangle\,|$ and $C(n,i)$ is n choose i. Because each test case has an equal probability of being in any permutation, this means that $T_i \in \langle T_1, \ldots, T_n\rangle$ could be executed $O\left(\frac{1}{n}\sum_1^{n-1} C(n,i)\,(n-i)!\right) = O\,(n!)$ times during GA execution. The cost of the fitness function could be greatly reduced by collecting coverage data on a per-test basis. Then the algorithm only needs to execute a given test case twice; once to get time data, and once to obtain coverage information. The fitness function then reduces to a calculation that merges the coverage data of the test cases under consideration. This will substantially improve performance without any impact on effectiveness.

As an example of the performance improvement attained by using per-test instead of aggregate coverage information, consider the worst case fitness calculation that could occur

in our case studies. Assume that a selection σ of `JDepend`'s test suite includes all 53 test cases. Although σ itself only requires about 5.5 seconds to execute, it would take about 80 seconds to calculate the fitness of σ using aggregate coverage information. However, if tool support is available to compute coverage information in a per-test manner, then a single merge of 53 coverage data files would require only about 0.13 seconds. Then, assuming no memoization, a fitness calculation for σ based on per-test data would require only about 3.5 seconds, which is an improvement of two orders of magnitude. With per-test data, no test cases need to be rerun for the fitness calculations during GA execution, and thus the time overhead depends only on the time required to perform $|\sigma| - 1$ merges.

Discussion. According to results in [114], the APFD values for `Gradebook` were similar regardless of the value that was used for (g_{max}, s). However, Figure 4.7 reveals that a change in (g_{max}, s) had a significant impact on the time overhead of time constrained test suite selection. It is also clear from Figures 4.6(c) and 4.7 that on average, block coverage outperformed method coverage in relation to APFD while not increasing the time overhead of test suite selection. Based on our empirical data, a configuration of GAPrioritize that uses $(g_{max}, s) = (75, 15)$ and $tc = block$ would yield the best results in the shortest time.

Even though the time required to perform test suite selection is greater than the execution time of the test suite itself when aggregate data is used, a given selection can be re-used each time a software application is changed. Selection reuse is typical in general test case prioritization and selection [89]. In this way, the cost of the initial selection is amortized over the period of time during which the selected test suite is used. Initial selection cost can also be greatly decreased if per-test coverage data is available. Even in light of the time required for selection, the empirical study suggests that it might be advantageous to use the presented technique when there is a fixed set of short testing time constraints. This is especially evident when time-constrained selections are compared to alternative non-constraint-aware prioritizations.

```
Algorithm FindAllSelections(T, limit, perm, tt)
Input:  Test suite T                          Total time allowed limit
        Current partial permutation perm      Time taken by perm tt
1.   scanned ← 0;
2.   for T_i ∈ ⟨T_1, ..., T_n⟩
3.       do if perm contains T_i
4.             then continue;
5.          if tt + time(⟨T_i⟩) < limit
6.             then scanned++;
7.                  add T_i to end of perm;
8.                  mark T_i as used;
9.                  FindAllSelections(T, limit, perm, tt + time(⟨T_i⟩));
10.                 remove T_i from end of perm;
11.                 mark T_i as not used;
12.  if scanned = 0
13.     then store(perm)
```

Figure 4.8: The Complete Selection Generator.

Alternative Comparisons

Random Selections. According to Do et al., randomly ordered test cases are useful because they redistribute fault-revealing test cases more evenly than original test orderings [27]. Using 18 computers, 10,000 selections were randomly created on each machine over several days. Three elements were varied to create the 18 configurations: (i) the percent of total test suite execution time p_t, (ii) number of faults $|\Phi|$, and (iii) the application P. A building approach was used to create the test tuples, as shown in Figure 4.8. For each selection, a test case is chosen, where test cases are sequentially checked for previous use. If the test case fits in the new permutation, then it is added, and the algorithm recursively analyzes the new tuple, as on line 9. Selected test cases are added until the next tuple to be added causes the test tuple to exceed t_{max}. Each of the generated selections nearly fills the time limit but does not go over that limit.

This algorithm generates all possible test tuples that will fit in the allotted time. The *store(perm)* method on line 13 stores each tuple for later use. From these saved tuples, 10,000 selections were randomly selected. Alternatively, each random selection could be

built by incrementally adding random unused test cases to the tuple until the time limit is reached.

Success of the genetic algorithm selections is measured by comparing the selected test suites' APFD values to the APFD values of the other reorderings. Figure 4.9 shows a comparison between the APFD values, percent of total test suite execution time, and the number of faults seeded for GA-produced selections in relation to randomly produced permutations. Figure 4.9(a) describes the results for `Gradebook`, and Figure 4.9(b) does the same for `JDepend`. Each bar in the graphs represents the average of the APFD values of 10,000 random selections, and the error bar shows the standard deviation from the mean APFD.

In the case of `Gradebook`, the GA-produced selections performed extremely well in comparison to the randomly produced selections. All APFD values from selections based on the `Gradebook` application were more than one standard deviation above the mean of the randomly produced selections. Because the tests that detect the most faults in `Gradebook` are longer in execution time and fewer in number with regard to the other test cases, there was a greater probability of creating weak test tuples using random selection. As depicted in Figure 4.9(a), the test tuples executing with $p_t = 0.25$ had negative APFD values on average because they were only able to find a few of the seeded faults. Thus, there is a clear benefit to using intelligently prioritized tests instead of random selections.

In the case of `JDepend`, the GA-produced selections on average did not perform as well as the selections of the test suite for `Gradebook`. This was anticipated because of the nature of `JDepend`'s test cases, which are much more interchangeable with respect to fault detection potential than those of `Gradebook`. As can be seen in Figure 4.9(b), on average, all GA-produced selections that ran within 25% of the total test suite execution time had APFD values more than one standard deviation above the mean APFD value of the same set of randomly produced selections. GA-produced selections that ran within 50% and 75% of the total test suite execution time also had APFD values within one standard deviation above the mean of the randomly produced selections.

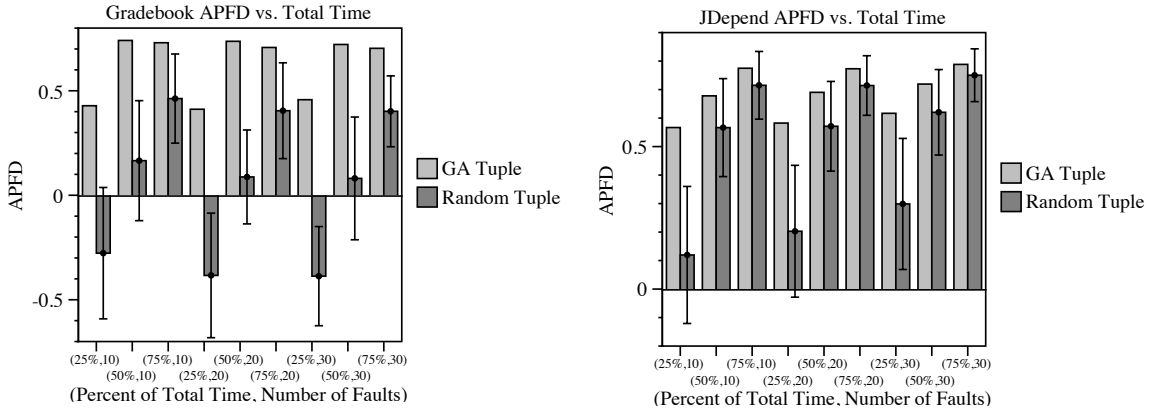


Figure 4.9: GA vs. Random Ordering APFD Values.

Because the test cases of `JDepend` all have about the same adequacy and take around the same amount of time to execute, many different test subtuples have the same APFD. As observed in Figure 4.9(b), the average APFD for test tuples that are allowed to run in 75% of the total test suite execution time is likely to be closer to the best possible APFD value than that of test subtuples that are allowed to run in only 25% of the total test suite execution time. In other words, it is much easier for random selections to have high APFD values when more of the original test suite can be run, particularly in the case of `JDepend`.

Overall, the GA-produced selections performed extremely well in comparison to randomly generated selections. Nearly all results were more than one standard deviation above the mean APFD values calculated for selections that were produced randomly. All results had APFD values that were greater than the mean APFD values of random selections. Note also from Figures 4.9(a) and 4.9(b) that APFD values for the percent of total test suite execution time groups are all very similar. This again provides confidence in the results generated by the genetic algorithm because about the same percentage of faults can be found by any of the selections in spite of how many defects there are or how the GA created the selections.

Additional Selection Techniques. Two simple forms of selection include those

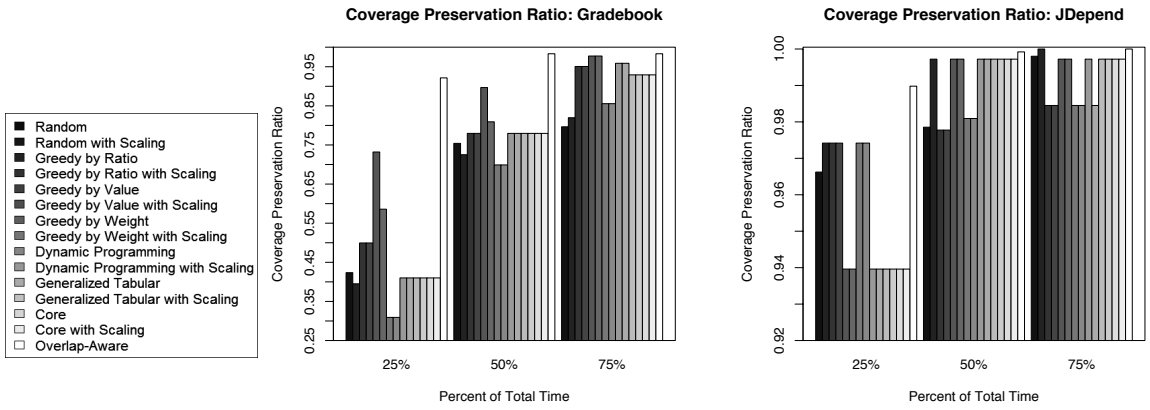
		Initial	Reverse	Fault Aware	GA	Initial	Reverse	Fault Aware	GA
p_t	$\|\Phi\|$	`Gradebook`	`Gradebook`	`Gradebook`	`Gradebook`	`JDepend`	`JDepend`	`JDepend`	`JDepend`
0.25	10	-0.600	-0.233	0.660	0.428	0.525	-0.300	-0.050	0.567
0.25	20	-0.863	-0.208	0.720	0.412	0.478	-0.275	0.050	0.649
0.25	30	-0.892	-0.006	0.453	0.457	0.473	-0.133	0.083	0.617
0.50	10	-0.042	0.160	0.869	0.741	0.873	0.000	0.200	0.678
0.50	20	-0.192	0.167	0.873	0.737	0.819	0.013	0.175	0.690
0.50	30	-0.308	0.284	0.782	0.722	0.842	0.100	0.208	0.719
0.75	10	0.314	0.478	0.906	0.730	0.878	0.492	0.590	0.775
0.75	20	0.124	0.433	0.926	0.707	0.826	0.608	0.283	0.773
0.75	30	0.049	0.516	0.880	0.703	0.848	0.534	0.250	0.788

Table 4.5: Initial, Reverse, Fault-Aware, and Genetic Algorithm Selection APFD Values.

that execute test cases in the order in which they are written or the reverse of that order. Table 4.5 compares GA-produced selections to initial and reverse ordering selections. The genetic algorithm produced selections that were up to a 120% improvement over initial orderings. For example, `Gradebook`'s initial tuple created using $p_t = 0.25$ and $|\Phi| = 30$ had APFD $= -0.892$ whereas the associated intelligently produced tuple had APFD $= 0.457$, as shown in Table 4.5. The time constrained selections were also an improvement over all reverse ordering selections in both `JDepend` and `Gradebook`.

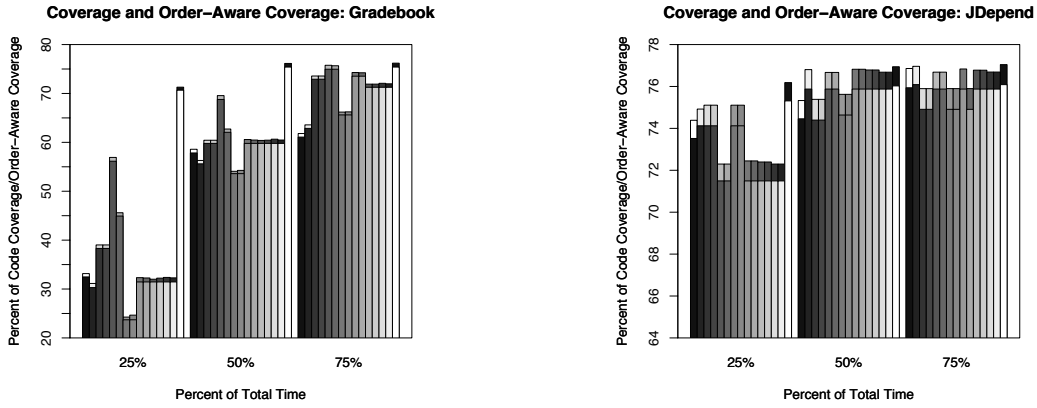
Even though fault-aware selection cannot be performed in practice, these reorderings are useful for comparison purposes. The fault-aware selections were constructed by first calculating the time required and the faults detected by each test case. Then, in a non-overlap-aware manner, the tests are added to the tuple based on the fault/time ratio until the addition of the next test would exceed the time limit. The `JDepend` GA-produced test tuples performed much better than the fault-aware selections described in Table 4.5. This is likely because most of the test cases in `JDepend` cover the same code segments. While the genetic algorithm identifies the overlap in test code coverage (and thus the fault detection potential), the fault-aware selection does not. Thus, the GA produced markedly better results for `JDepend`.

On the other hand, the selections produced by the GA for `Gradebook` were not quite as good at finding defects quickly when compared to the fault-aware selections for `Gradebook`, as noted in Table 4.5. This is because `Gradebook`'s test cases have little coverage overlap, causing few test cases to detect the same faults. Because the fault-aware selection technique



(a) (b)

Figure 4.10: Coverage Preservation of Test Suite Selection.



(a) (b)

Figure 4.11: Genetic Algorithm Comparison to Non-Overlap-Aware 0/1 Knapsack Solvers.

has actual knowledge of all faults, it could specifically organize the test cases to best find the known faults without concern for fault detection overlap. Although the genetic algorithm's results did not have as high of APFD values in this case, its selections are more general because they are not based on specific faults. Thus, they have the potential to perform well no matter where the defects in the code may exist.

0/1 Knapsack Selection Comparisons

Finally, we compare our genetic algorithm's results to the selections produced using the non-overlap aware 0/1 Knapsack solvers that were presented in Chapter 3. In these experiments, block coverage was used. Here we examine the overall coverage, order-aware coverage, and coverage preservation of each of the resulting selections. These can be seen in Figures 4.10(a)- 4.11(b). As would be expected, the coverage overlap-aware solver achieves the highest overall coverage for each testing time constraint. We learn from each of the figures that the GA approach is particularly effective when creating selections for execution times that are much smaller than the execution time of the original test suite. The most sophisticated non-overlap-aware 0/1 knapsack solvers do not always create the most effective selection of the test suite, suggesting that overlap-aware selection techniques, such as GASelect, that have a higher time overhead are appropriate in contexts where correctness is the highest priority.

4.4 Conclusions

In this chapter, we have described a search-based technique that selects test cases for time constrained execution. When provided with a testing time budget, the GASelect algorithm evolves a reordered test suite that rapidly covers the test requirements and always terminates within the specified time limit. Experimental analysis demonstrates that our approach can create time-aware selections that significantly outperform other non-time-aware prioritization techniques and non-overlap-aware selection techniques. In one example, our technique created selections that, on average, had up to a 120% improvement in APFD over other prioritizations. Even when there is a 75% reduction in the time available for test execution, our approach preserves 94% of the original test suite's code coverage. The empirical study also reveals methods for improving the performance of time-aware selection. For instance, coupling a reduced population size with an increase in the number of generations decreases selection time by up to 43% without compromising the effectiveness of the

resulting selection. We also evaluate the performance benefits of using per-test coverage data instead of aggregate data. Taking advantage of per-test information allows for a substantial speedup of the fitness function calculation. In the worst case for our applications, this can result in up to a performance improvement of about two orders of magnitude with no impact on effectiveness.

In confirmation of the predominate understanding of search-based methods in software engineering [40], the experimental results suggest that GASelect incurs a high computational cost, and thus it is most valuable when the resulting prioritization is re-used multiple times. The empirical study shows that calculation of $F_{s-actual}$ and consideration of coverage overlap improves the effectiveness of the prioritized test suite; however, it also increases the time overhead of selection. Calculating fitness was expensive because *fit* accommodates the use of a test coverage monitor (e.g., Clover [53], Jazz [76], and Emma [91]) that only reports aggregate coverage data. However, our analysis shows that if a per-test coverage calculator is available, the overall time overhead can be much reduced.

In the next chapter, we consider the challenge of developing a test execution system that can be used to accurately and efficiently evaluate the quality of our test suites.

Chapter 5

Executing Test Cases for Branch Monitoring

Contents

1.1 Test Case Selection **2**

1.2 Executing Test Cases **4**

1.3 Constraints During Testing **4**

1.3.1 Constraints When Selecting Test Cases 5

1.3.2 Constraints When Executing Test Cases 7

1.4 Challenges and Goals of Testing in Resource-Constrained Environments **8**

1.5 Research Overview **11**

1.5.1 Description of the Research Process 12

1.5.2 Contributions of the Dissertation 15

After tests are generated and selected, test execution is monitored to determine the quality of the tests being run. While there are many existing tools that evaluate test suites during execution, nearly all suffer from high time overheads and even larger memory overheads due to their reliance on software-level instrumentation. The high overheads incurred make existing monitoring tools difficult to use within resource-constrained environments.

As an alternative to instrumentation-based monitoring, in this chapter, we explore the potential of exploiting hardware mechanisms for monitoring in software testing. Modern-day processors include sets of hardware performance mechanisms that were designed to

track and measure various aspects of program and kernel execution. For example, the Intel Nehalem processor provides the capability to track more than 2000 different performance events, and recent Linux kernel patches provide user-level support for nearly 200 of these mechanisms [31]. Hardware performance counters (HPCs) are counters for such events, stored in registers and accessible using privileged instructions. HPCs count events such as the number of instructions retired, cache misses, and branches executed. Hardware performance monitors (HPMs) are a related type of mechanism that stores information about events, such as addresses from executing instructions. To extend the capabilities of HPCs and HPMs, newer mechanisms have been developed to buffer or filter data, as well as providing control over data caching and instruction reordering. For generality, we will use the term hardware mechanism to include HPCs, HPMs, and mechanisms throughout the rest of this dissertation.

In this chapter, we explore how hardware performance mechanisms can be used for branch testing. As described in Chapter 1, hardware mechanisms can be used for sampling program execution with very little overhead; their use can potentially remove the need for instrumentation entirely.

Most hardware mechanisms are designed to monitor performance. The majority were introduced for tasks such as system tuning or compiler optimization improvements. More recently, additional mechanisms have been added as debug facilities for use in debugging application software, system software, and multitasking operating systems [47]. Although all modern processors have hardware mechanisms, the list of these differs between processors. For many years, these counters and mechanisms were only accessible through the direct use of privileged instructions. In the past 5 years though, support for many of the more popular mechanisms has been provided at the user level through interfaces and helper libraries. However, newer mechanisms, such as those in the debug facilities, still are not supported at the user level.

Hardware mechanisms that have been made visible at the user level have been leveraged very successfully in software systems because of their low overhead monitoring capability.

Some systems that exploit hardware monitoring are discussed in Chapter 2.2.3. However, hardware counters and mechanisms have yet to be exploited in branch testing.

In this chapter, the potential of adapting hardware performance mechanisms for use in test execution is analyzed. We focus on branch coverage evaluation because a number of more advanced hardware mechanisms directly monitor branch execution. The first mechanism, a performance-monitoring feature available in all recent Intel processors, is the last branch record (LBR). The LBR stores the origin and destination addresses of all taken branches in a circular buffer of special-purpose CPU registers. An interrupt enables the information in the LBR to be read.

The Branch Trace Store (BTS) mechanism is another potential tool for use in branch testing because it records all branch information automatically in a large in-memory buffer. When the buffer is filled, an interrupt is generated to allow the kernel to retrieve the recorded data and reset the buffer pointer so that no branch information is lost.

In this chapter, we analyze the tradeoffs between gathering executed branch information from the LBR versus the BTS in terms of overhead and effectiveness for branch testing. Although the LBR and BTS are both capable of monitoring and reporting all taken branches, the LBR is intended to sample branches while the BTS is meant for branch tracing because of the structures used to record branches.

This work represents the first attempt to explore how hardware performance mechanisms can be used for branch testing. The goal is to develop an approach that exploits hardware monitoring for branch testing with low overhead and high precision. We first demonstrate the costs associated with accumulating a complete branch trace generated by sampling the BTS. We then examine a traditional sampling technique's effects on branch monitoring in terms of time and effectiveness for branch testing using the LBR. Because the LBR and BTS are unable to inherently observe fall-through branch edges, innocuous unconditional branches are placed along fall-through edges and thus can be detected by the hardware mechanisms. To evaluate how the precision of branch testing through hardware sampling can be improved, we also explain and develop sampling techniques to mitigate the threat of

sampling bias, reduce the frequency of monitoring repetitious and unimportant branches, and increase overall sampling effectiveness through the use of multiple cores.

This chapter provides empirical evidence that a high rate of precision can be maintained with low memory overhead through the use of the LBR. Gathering a complete branch trace using the BTS, however, is prohibitively expensive for branch testing due to its design as a debugging tool. When monitoring using the LBR, we learn that a simple event-based sampling technique is fairly effective for branch monitoring at low cost. It is especially useful when memory overhead is a concern. We demonstrate that on average, our memory overhead is only increased by 0.57%, which is much improved over instrumentation techniques whose memory overheads can be prohibitive. A high percentage of actual coverage can be attained using event-based sampling techniques, but at a cost higher than instrumentation. However, we also demonstrate that once LBR filtering mechanisms are made visible at the user level, the time overhead of sampling the LBR can likely be reduced by more than 10% compared to using full instrumentation and can attain approximately 65% of the actual branch coverage. Thus, the LBR has the potential of being extremely useful for branch testing purposes.

5.1 Challenges of Exploiting Hardware Mechanisms

There are a number of challenges in using hardware mechanisms for branch coverage evaluation during test execution. These challenges are due to the differences between software-level instrumentation versus sampling execution at the hardware level.

Typically with branch testing, instrumentation can precisely monitor all branches in a program's source code. To obtain a complete set of taken branches, the BTS should be used. However, the time overhead of acquiring a full trace is likely to be high because of the cost of writing to memory on every taken branch. To reduce the cost, the LBR instead can be sampled at intervals. While a low-cost acquisition of full branch information using the LBR is unlikely due to the cost of interrupts, intelligent sampling techniques can

be used to significantly reduce the overhead of branch monitoring while maintaining high branch coverage. This work analyzes the precision that can be obtained using hardware mechanisms and the tradeoffs between the precision and cost of sampling the hardware.

In addition to the tradeoffs between maintaining low overhead while acquiring a high level of branch coverage, the application of hardware mechanisms to branch testing inherently presents other challenges. One of the main challenges involves the amount of branch information that is observed. In structural testing, we are concerned with gathering branch information only for branches in the source code. The LBR and BTS, however, monitor ALL taken branches executed on the system. On some processors such as the Intel Nehalem family, it is possible to filter the branches that are collected based on the branch type or privilege level, but user level support for filtering is not yet available for the LBR. Privilege level filtering is supported for the BTS.

An additional challenge is related to the precision of reported branch information. When performing sampling using the LBR, a performance counter is configured to count the number of executed branches and to generate an interrupt when the specified number of branches has been observed. At that point, the values in the LBR are read. The BTS buffer is similarly sampled whenever the BTS threshold counter overflows signifying that the buffer is nearly full. In both cases, because of latency in the microarchitecture between the generation of events and the generation of interrupts on overflow, it is sometimes difficult to generate an interrupt close to an event that caused it. Thus, some information may always be missed [47].

A third challenge arises from the fact that the hardware is monitoring the execution of binary code. Instrumentation monitors on a source code level and thus tracks both taken and fall-through sides of a branch. Hardware that monitors branches, however, only can detect branches based on a jump from a source to some target. Thus, fall-through paths are not recorded by branch monitors. A supplementary technique is needed to account for both edges of a branch.

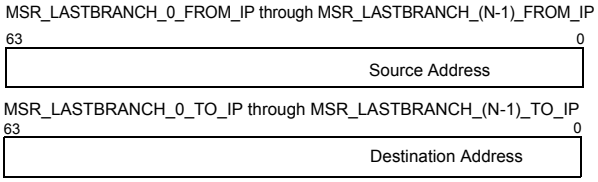


Figure 5.1: 64-bit Layout of the LBR MSR [47].

5.2 Hardware Monitoring for Branch Testing

Beginning with the Intel P6 family of processors, breakpoints can be set on taken branches, interrupts, and exceptions, and single-stepping from one branch to the next is possible for the purposes of debugging and profiling. The Intel P6 also has the ability to log branch trace messages in memory. These features have since been extended to other Intel processors such as the Pentium 4, Core Duo, and Core i7. Such enhancements enabled the creation of the LBR and BTS mechanisms.

5.2.1 Last Branch Record (LBR)

The LBR was intended as a profiling tool for sampling partial branch paths in the operating system. The LBR branch vector of registers is available in many processors, and the number of branches that the LBR can hold is increasing with each new processor family [47]. When the LBR is turned on, the processor records a running trace of the most recent branches, interrupts, and exceptions taken by the processor. Each branch edge is represented as a source and destination address and is stored into a pair of LBR registers, such as the one pictured in Figure 5.1

The LBR is made up of a circular buffer of n LBR model-specific registers, where $n \geq 4$. In current Intel processors such as the Nehalem i7, $n = 16$. This means that at any sample point, the last 16 executed branches can be recorded from the LBR. This set of n correlated branch events that represent a partial path of program execution define a branch vector.

The LBR can be sampled whenever an interrupt is generated. An interrupt occurs when

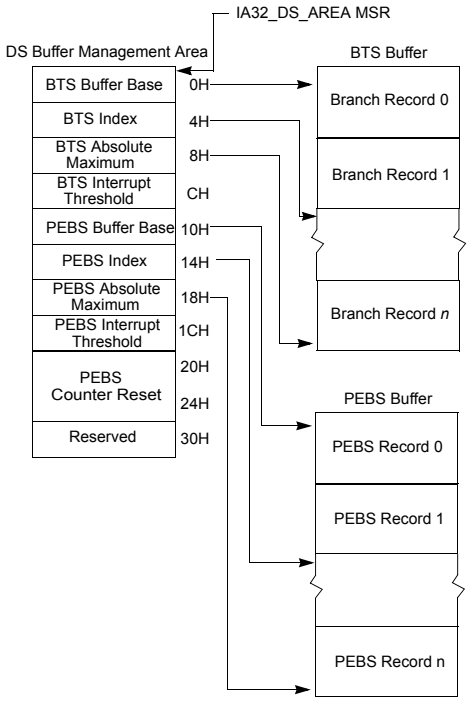


Figure 5.2: The Debug Store Area [47].

a performance counter detects an overflow from a hardware counter. Common hardware counters include cpu-cycles, retired branches, retired instructions, etc. The user must specify the rate of sampling based on these simple counters. When the interrupt is generated, the LBR branch vector can be polled for branch data, and the information can be processed.

Because the LBR is filled with ALL taken branches, interrupts, and exceptions that occur on the processor, some newer processors such as those based on the Intel Nehalem microarchitecture provide an additional LBR filtering mechanism. Filters weed out branches prior to placing them in the buffer based on their privilege level or certain branch type conditions. When LBR filtering is enabled, the LBR stack only captures a subset of all branches. Unfortunately, LBR filtering is not currently supported at the user level.

5.2.2 Branch Trace Store (BTS)

The BTS was developed as part of the processor's debugging facility and makes up half of the Debug Store mechanism, as can be seen in Figure 5.2. The BTS works in conjunction with the LBR by requesting that the LBR send each branch record out on the system bus in addition to recording it. The use of the BTS is appealing because it records and reports all executed branches in the order in which the branches occur without the need of frequent sampling. However, the BTS design is much different than that of other hardware mechanisms. The BTS sends each executed branch to a buffer, but it also clears the instruction pipeline on every branch in order to maintain correct branch ordering.

Because of this design, recording all executed branches using the BTS can greatly reduce the performance of the processor. In efforts to reduce this cost, the BTS has its own set of filters based on branch privilege level. User level or kernel level branches can be filtered out, in which case the filtered branch records are not sent out on the system bus or logged. These filtering mechanisms are visible in user space.

The BTS mechanism additionally provides the capability of saving the branch source and destination addresses in a memory-resident BTS buffer, which is part of the Debug Store save area. The BTS buffer can be configured to be circular so that the most recent branch records are always available, or it can be configured to generate an interrupt when the buffer is nearly full so that all the branch records can be saved.

5.3 Sampling Hardware Mechanisms for Branch Testing

To achieve the goal of exploiting hardware mechanisms for branch testing purposes, sampling of the LBR and BTS must be performed in a way that produces low overhead while maintaining a high degree of source code-level branch coverage. Because the BTS incurs high overhead by simply turning it on, independent of sampling technique, we focus on using it only to acquire a full branch trace. Thus sampling of the BTS only occurs when the BTS buffer is nearly full. The LBR, on the other hand, can be sampled using any sample period,

with cost rising as the number of interrupts increases. More frequent sampling, however, also increases the precision of our sampling technique because more branches are observed.

We first take a traditional event-based sampling approach to sampling the LBR using a set of sampling periods. In attempt to improve the precision of this sampling technique, each of the challenges discussed in Section 5.1, which are inherent issues for sampling both the LBR and BTS, are addressed. Finally, the potential of using multiple cores to enhance sampling results is examined.

5.3.1 Event-Based Sampling

Event-based sampling is the standard technique used to monitor the LBR. In event-based sampling, a performance counter that monitors executed branches is configured to generate an interrupt when the counter overflows. To trigger an overflow, the counter is preset to a modulus value that will cause the counter to overflow after a specific number of branches have been executed. When the counter overflows, the processor generates an interrupt. The interrupt service routine then records the return instruction pointer, resets the modulus, and restarts the counter [47].

As an example, if the sampling period for the LBR is set to five thousand, an interrupt will be generated every five thousand branches. At that point, the LBR is polled and the branch vector is processed. The counter is then reset to zero, and LBR sampling continues. The sampling period can be set to any value desired. However, as branch information is more frequently sampled, the overall cost of monitoring branch records also rises.

5.3.2 Addressing the Challenges of Sampling

Event-based sampling is influenced by several factors that can reduce the effectiveness of monitoring coverage information for branch testing. These include the inability of the LBR to see fall-through branch edges, the sampling bias caused by the synchronization of the executing program with the sample period, and the large amount of extraneous and

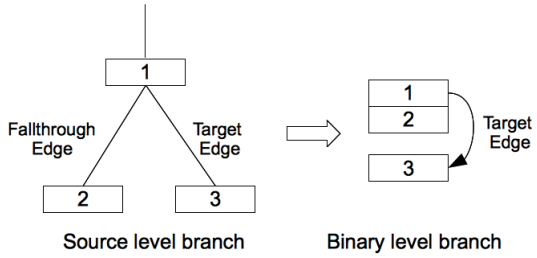


Figure 5.3: The LBR is incapable of detecting the fall-through branch edge from 1 to 2.

repetitious data that the LBR observes. Because the BTS acquires its branch records from the LBR, sampling the BTS presents the same challenges.

Enabling Fallthrough Observation

Independent of sampling technique, a leading source of low coverage monitoring effectiveness is due to the fact that the LBR alone cannot observe when fall-through branches have occurred. In branch testing, a tester wants to ensure that both edges are taken through a branch. For example, in Figure 5.3, monitoring should be able to detect both the execution of the fall-through path from 1 to 2 and the target path from 1 to 3. While this is obvious when looking at a flow graph, in the binary code, a branch is made up of some kind of jump to a target followed by another instruction. The LBR will report the jump from 1 to 3 but not the fall-through from 1 to 2. Therefore, the LBR by itself is only capable of monitoring 50% of the source level branches.

Fallthrough observation is possible using hardware monitoring by supplementing LBR monitoring with other event data. For example, the `INST_RETIRED` event could be polled in addition to the LBR to look for fall-through instruction execution. However, because this research focuses on the capabilities of the LBR, we instead give the LBR the potential to observe the fall-through path by inserting harmless unconditional branches along every fall-through edge. Insertion of the branches is automated through the use of a simple **program modification** tool that causes little additional execution time and nearly negligible code size increase.

Mitigating Sampling Bias

Also reducing branch sampling precision, sampling bias is a common occurrence in hardware sampling. This is particularly a problem when sampling events that occur often, such as branches. The sampling period can be in lockstep with the execution of the monitoring program, causing some branches to never be captured even though they are executed very frequently [56]. As an example of sampling bias, consider the execution of a benchmark called *mcf* where the sampling period is set to n, which is the number of branches in the LBR. The actual branch coverage obtained from running *mcf* is 72.37%. However, due to branch records being lost near interrupt points, the LBR reports only 69.23% branch coverage. Because the performance monitoring unit is designed for statistical sampling, **randomization** can be used to improve the accuracy of samples.

Randomization is principally useful when there is branch repetition. This generally happens when monitoring inside a loop. Randomization otherwise would have little effect on overall coverage observed. Randomization could also be helpful if branch information is aggregated over multiple program executions using factors of a particular sampling period. For instance, if one execution is monitored using a sample rate of 10000, other executions using periods of 2000, 5000, or 10000, for example, should be randomized at the repeated sampling points in order to improve precision.

Managing Extraneous Branch Samples

When performing traditional event-based sampling of the LBR, our preliminary work demonstrated that there are a large number of polling periods that produce branch vectors in which few or no branches are associated with the program's source code. For example, when sampling the application *bzip2* every 50,000 branches, interrupts, and exceptions, although nearly 50% of the actual branch coverage is observed, approximately 20% of the 13 million branches sampled contain no branch information related to the source code.

These sampled branches may be instructions that we do not monitor in branch testing such as unconditional branches, calls, exceptions, or interrupts. Also, some may be branches

executing at the kernel level. None of these need to be monitored in branch testing. Thus, in light of these observations, a **filtering mechanism** based on privilege level and types of branches seen is necessary to efficiently collect branches from the LBR for branch testing. Such a filter is available in some processors, but no interface is currently available for kernel or user access. A privilege level filter mechanism and user interface are available for the BTS.

5.3.3 Improving Effectiveness with MultiCores

To further improve branch sampling effectiveness, multiple cores can be used to increase the amount of sampled branches while maintaining a low overhead. Most commodity machines bought today include processors with multiple cores, and each core can be used to perform additional monitoring of the program under test. A core can sample its LBR at any rate in parallel with the others. By combining and shifting sampling periods across cores, the potential of observing unique branches grows with small additional cost.

5.4 Empirical Evaluation

The primary goal of this chapter's empirical study is to evaluate the adaptability of the LBR and the BTS for branch testing purposes. In addition to demonstrating the effectiveness and efficiency of performing branch testing by sampling the LBR and BTS for monitoring on single and multiple cores, we also implement possible solutions to each of the challenges that using the LBR, and by association the BTS, presents. The goals of the experiments are as follows:

1. Evaluate the time overhead and code size increase introduced to modified programs that allow the LBR and BTS to observe fall-through branch paths.

2. Demonstrate the overhead incurred by sampling the BTS for branch tracing.

3. Analyze the trade-offs between efficiency and precision of code coverage calculation using traditional event-based sampling of the LBR over multiple periods.

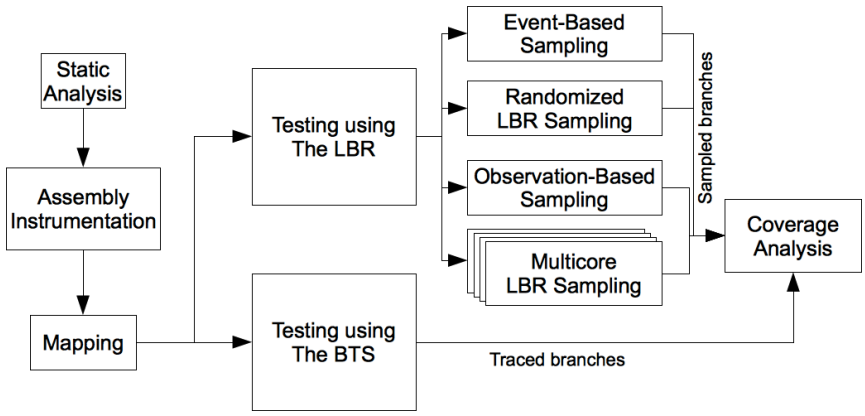


Figure 5.4: Overview of infrastructure to adapt LBR monitoring to branch testing.

4. Determine if a multicore approach to calculating coverage can significantly improve precision.

5. Observe the effects of sampling bias due to synchronization problems on branch testing.

6. Demonstrate the potential improvements on coverage and efficiency through the use of branch filtering

5.4.1 Experiment Design

All experiments were performed on a Intel Core i7 860 / 2.8 GHz quad-core machine with 4GB of memory running Linux Kernel 2.6.32. The Intel Core i7 processor was selected because it is part of the Nehalem family, which has a LBR buffer that contains the most recent 16 taken branches executed. Filtering is supported at a hardware level on the Nehalem for both the LBR and BTS.

Although there are a number of APIs available for taking advantage of performance monitoring hardware including OProfile [64], PAPI [19], and Perfmon2 [31], none of these yet support LBR or BTS monitoring. Perfmon is supported by a user-level tool, libpfm, and a kernel-level interface, perfevents [31]. However, because Perfmon has not been updated

to to take advantage of libpfm4, the most recent helper library to perfevents, we use and modify libpfm4 and perfevents directly. The libpfm4 library helps encode performance events to use with the operating system kernel's performance monitoring interface, and it contains all of the Performance Monitoring Unit (PMU) model-specific information such as the events names and encodings, and the various constraints between events. It is one of the most robust and flexible PMU interfaces and supports a wide range of microarchitectures.

While the current perfevents and libpfm4 do not provide an interface to the LBR, we were able to modify perfevents at the kernel level to include LBR support using a proposed patch [31]. The LBR is accessed through a new `PERF_SAMPLE_BRANCH_STACK` sample type. This allows for sampling of all taken branches without any filtering capabilities. An addition kernel patch would be necessary in order to access the Nehalen's filtering abilities. Because filtering is not used, the LBR records all branches, interrupts, and exceptions at both user and kernel level. No patch yet exists to enable filtering. We also patched libpfm4 to enable the setup and polling of the LBR. All sampling techniques were implemented into the libpfm4 package. Libpfm4 and perfevents already support BTS monitoring with filtering.

An overview of our testing infrastructure is shown in Figure 5.4. Before executing and monitoring a program, a simple static analysis is first used to identify and store the branch edges in the program's source code. The branch edges are stored in a hash table along with information pertaining to the associated source code lines, obtained from gcc's debugging information. This table is used as a checklist of branches with which we are concerned and is later used to calculate overall branch coverage. Next, each of the benchmarks is instrumented to enable the LBR to monitor fall-through branch edges as well as taken branch edges. The program modification process is explained in Section 5.4.2. All modifications are annotated into the branch hash table.

In the next step, the program is executed while the LBR or BTS performs monitoring. The BTS is configured to generate an interrupt when the buffer is nearly full, thus building a complete branch trace as it is sampled. When the BTS buffer threshold overflows, all branches in the BTS buffer are processed. Similarly, when the branches executed counter

overflows because it has reached the desired sampling period, the 16-branch branch vector in the LBR registers is read. Each branch is checked against the hash table of source code-level branch edges. If a branch is found in the hash table, the branch is marked as having been taken.

Once the program under test has finished executing, the sampled branch coverage is calculated based on the number of source code-level branches observed divided by the total number of source code-level branches in the program.

Benchmarks

Our experiments were carried out using the SPEC2006 C Integer Benchmarks. This set is made up of nine programs, although we do not report results for `perlbench` or `gcc` because both throw exceptions during normal execution on our system. Each program was compiled with debugging information and with no optimization options specified. Execution of the benchmarks was tied to a single core for all sampling techniques other than the experiment that uses multiple cores. In the multicore technique, a copy of the benchmark is executed and tied to each individual core. Each benchmark was run three times on the SPEC *test* set of inputs. To reduce the risk of tainting from other processes, all experiments were executed after all other possible user processes were killed.

Metrics

To achieve our experiment goals, three metrics are considered: the percentage of actual coverage, time overhead, and code size.

In order to evaluate the effectiveness of our sampling techniques, we first calculate the total number of branches that exist in the benchmark's source code, *total*. We then iterate through the hash table and calculate how many of those branches were observed by the LBR, LBR_total. For comparison, we also calculate the actual number of branches covered, $actual\_total$, using Pin, a commercial software-based dynamic binary instrumen-

Benchmark	Branch Edges	Real Branch Cov.
bzip2	2154	56.36%
gobmk	27558	50.39%
h264ref	14674	24.37%
hmmer	8830	6.55%
libquantum	752	32.85%
mcf	362	72.38%
sjeng	5252	42.99%

Table 5.1: Number of branch edges and actual branch coverage of original program calculated using a software based instrumentation tool.

tation tool [69]. The *LBR observed coverage*, $\frac{LBR\_total}{total}$ is compared to the actual coverage, $\frac{actual\_total}{total}$ to determine the *percent of actual coverage* that the LBR achieves.

The efficiency of the sampling techniques and the time overhead incurred from modifying the program to enable fall-through observation is calculated based on the base run times of benchmark execution reported by the execution tool of the SPEC2006 benchmarks, `runspec`. All timing comparisons are made to the overheads observed from execution of full software-instrumented versions of the benchmarks.

Code size measurements are taken using linux's du utility. The increase in the code size of modified programs is calculated by comparing the size of the original binaries to the size of the modified binaries and to the size of the software-instrumented binaries.

5.4.2 Experiments and Results

Experiments were run in order to analyze the time overhead and percent of actual coverage that can be achieved by sampling the LBR and BTS. We first evaluate the potential of applying the BTS to branch testing to evaluate its applicability. We then apply our program modification tool to enable the LBR to observe fall-through branches and analyze the time and memory overhead that it introduces. The modified programs are then monitored using three LBR sampling techniques to demonstrate the trade-offs between efficiency and precision of code coverage calculation using traditional event-based sampling of the LBR. Finally, we create a simulation of the LBR hardware filtering mechanism to evaluate its

Benchmark	Exec. Time (s)	BTS Time(s)	Overhead
astar	9.49	238	25X
dealII	14.9	865	58X
mcf	5.82	140	24X
milc	2.43	34.6	14.2X

Table 5.2: Observed BTS overhead on a subset of the SPEC2006 benchmarks.

potential success.

BTS Tracing

Because the BTS promises a certain time and space overhead just by turning it on, we start our experiments by tracing taken branches only using the BTS to gauge its potential. A buffer of size of 2 pages is used, and to minimize data loss, the threshold is set to overflow after 1 page. Because BTS filtering is supported at the user level, we constrain logged branch information to non-kernel level branches only. Branch type cannot be filtered, so unconditional jumps, calls, exceptions, and other branches with which we are not concerned will still be monitored.

A subset of our results can be seen in Table 5.2. We discovered that using the BTS generated time overheads averaging 40X with the lowest being 2X and the highest being 90X compared to native execution. Increasing the buffer and threshold sizes had negligible effects on the time overhead of the branch tracing. The extremely high time overhead is due to the way the BTS mechanism is implemented in hardware. Specifically, the cost is an effect of the trace store occurring on every taken branch. On each context switch, the BTS is disabled and reenabled, and the configuration is saved and restored in order to appropriately associate instruction pointers that are part of the branch records with the corresponding process. Simply turning the BTS on without sampling any of the branch data can account for 25 to 30X overhead [47], even when using filtering and not tracking fall-through branches. While these high time overheads may be acceptable for debugging, for the purposes of branch testing, the BTS overheads are inherently prohibitively high.

Benchmark	Native Time (s)	Mod. Time (s)	Instr. Time (s)	Native Size (kB)	Mod. Size (kB)	Instr. Size (kB)
bzip2	16.5	16.9	18.6	260	264	392
gobmk	30.8	33.1	39.8	8184	8204	9392
h264ref	43.8	43.8	45.9	2892	2912	3568
hmmer	11.2	11.4	11.8	1360	1372	1832
libquantum	0.155	0.162	0.166	208	208	260
mcf	3.66	3.86	4.03	128	128	156
sjeng	6.92	7.74	8.96	592	596	852

Table 5.3: Time overheads & code size of native, fall-through enabled, and software-instrumented benchmarks using *test* inputs.

Program Modification Overhead

Before evaluating sampling techniques for the LBR, we first analyze the time overhead and code size increase caused by our program modifying tool. Because not monitoring fall-through branch edges decreases the actual coverage that the LBR can observe by 50%, our fall-through enabling modification tool is applied to all benchmarks prior to sampling.

To enable fall-through branch coverage monitoring through the LBR, our tool first compiled each benchmark down to assembly code. Each instruction in the program is examined, and if the instruction is a conditional branch, a `jmp 9f; 9:` is added immediately after it. This added instruction is an unconditional branch that can be seen by the LBR and simply jumps to the original fall-through instruction. Once these innocuous branches are added along each conditional branch fall-through path, the assembly code is compiled using gcc to generate new executables.

The cost of our program modifications is much less than typical software-based branch instrumentation techniques, which often range from 10 to 30% time overhead and 60 to 90% space overhead for branch testing [76,106]. Usually instrumentation probes are placed either within the conditional branch statement itself or at the destinations of the fall-through and taken branch edges. If the instrumentation is associated with the branch as a whole, its payload needs to mark that the branch was executed and whether it was true or false. If it is placed along a branch edge, the payload need only mark that the branch was taken. However, this technique requires double the number of probes to be added to the program

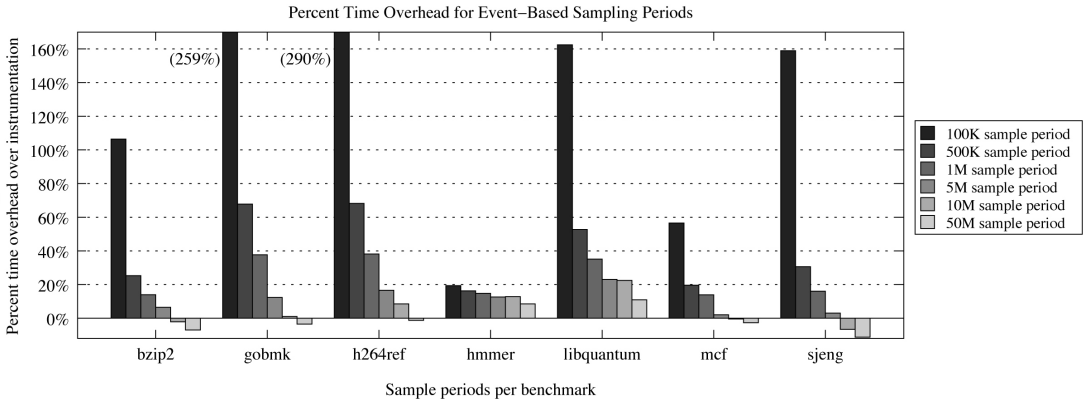


Figure 5.5: Time overhead for event-based sampling on a single core relative to full instrumentation.

under test [106].

We compare the time overhead and code size of the original program to 1) the program generated by applying our modification tool and 2) a fully software-instrumented program. We use TestCocoon to generate the instrumented programs [106]. The time and code size comparisons are listed in Table 5.3. On average, our modification tool generates a program with less than 5% time overhead and only 0.57% larger code size compared to native execution. Our modifications are much more lightweight than traditional instrumentation probes and payloads because ours consist of only unconditional jumps.

As this branch modification is the only contributor to increased memory overhead when sampling the LBR, we find that hardware monitoring techniques can be especially useful for testing in memory constrained environments. This is not the case for instrumentation. Full branch instrumentation of these 7 programs using TestCocoon results in time overhead ranging from 4.5 to 30% and code size increase ranging from 15 to 51%. On average, these values are low for full branch instrumentation, as is observed in related work [76] and in the TestCocoon documentation [106], but even the reported code size overheads could cause instrumentation to be inapplicable in certain settings.

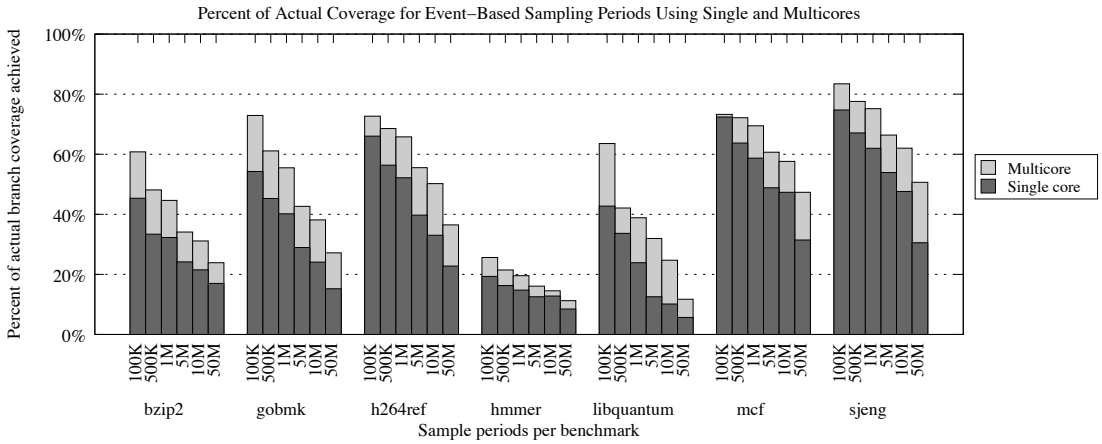


Figure 5.6: Percent of actual coverage from event-based sampling on single and multiple cores.

Event-Based Sampling

After a benchmark is modified to have observable fall-through paths, it is executed with the LBR turned on and configured to perform event-based sampling. Figures 5.5 and 5.6 show the effect of the sampling period on the runtime collection of branch vectors. The time overhead and percent of actual code coverage are shown for six sampling periods ranging from 1K to 5M relative to the time overhead incurred by performing full instrumentation using TestCocoon [106]. The interrupt thrown at the end of each sampling period is the main source of time overhead. As the sampling rate increases, the overhead also increases because more interrupts are thrown. Polling the LBR and processing the branches in the vector had negligible overhead.

As seen in Figure 5.6, the percent of actual coverage grows as the sampling rate increases, although at a much slower rate than the change in the time overhead. Despite that more samples are being processed, not all of these samples are associated with our source-code level branches, and of those that are, many have been seen and recorded before. The figures show that sampling with periods of 1 million results in an average of 24.2% time overhead, relative to instrumented code, with 40% of actual coverage for our benchmarks. However, some benchmarks perform much better. *Mcf*, for example, is able to achieve nearly 50% of

the actual coverage for 2% overhead over testing using instrumentation.

At smaller sampling rates, the percent of actual coverage is much improved. For a sampling rate of 1000, for example, 84.8% of the percent of actual coverage is observed. However, the time overhead required at this rate is prohibitively high. Other sampling techniques are therefore necessary to reduce the frequency of interrupts in order to improve the potential of applying LBR sampling to branch testing.

Randomized Sampling

The next set of experiments attempts to mitigate the threat of sampling bias. A kernel patch to enable LBR period randomization has been discussed, but it, like the LBR support, has yet to be accepted into the main kernel distribution. We implement our own version based on the discussion of the patch.

We perform randomized sampling by dynamically modifying the sampling period by very small amounts during monitoring. After each LBR polling event, a random number between 0 and 1 is selected. If the number is less than a user provided percentage, we use 5%, the period will be varied. The user provided percentage should be kept extremely low. Otherwise, the overhead of repeatedly changing the period will affect the time overhead of the LBR monitoring negatively.

If the rate is to change, a random 32-bit number is masked to keep the period within a maximum range of variation. The random number is then applied around the original period. Thus, on average, the sampling period remains equal to the initial period. We keep each period within 1% of the initial period.

Randomization is performed in order to mitigate sampling bias. However, because so many of the samples are extraneous, as discussed in Section 5.3.2, few are missed due to bias in our approach. Changing the period has a very small overhead, but, when performing it unnecessarily, repeated period change increases overhead while doing little to the overall branch coverage.

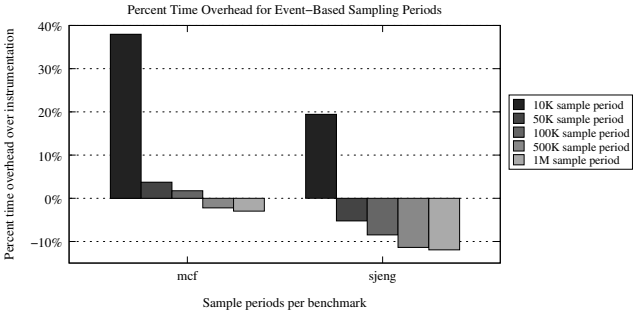


Figure 5.7: Time overhead relative to full instrumentation of a simulation of using a filtering mechanism.

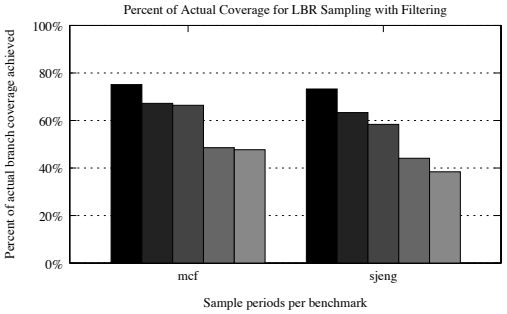


Figure 5.8: Percent of actual coverage obtained during a simulation of using a filtering mechanism.

Filtering Potential

Although filtering is not accessible at the user level yet, we next analyze the potential efficiency and effectiveness improvements that can be obtained through its use. We develop a simulation of a filter that mimics abilities of the LBR filtering mechanism, capturing only application level branches associated with source-code level branches. We ensure that no other user processes are being executed during LBR monitoring and assume for the sake of the experiment that sampling bias does not occur.

To perform this simulation, we first generate a trace of all branch edges taken during program execution. We then sample the trace, as the LBR does, by collecting the last 16 branches executed on each sample period. This allows us to calculate the potential percentage of actual coverage that can be obtained over a set of 5 sampling periods.

To estimate the time overhead of using such a filter in hardware, we calculate the number of samples m_p that would be taken for each sampling period p. We then execute the LBR on the benchmark using sampling period p, but stop the sampling mechanism once m_p samples have been taken. By using this technique, the actual times for LBR setup and teardown, sample processing, and performing the interrupts are incorporated.

Note that the coverage results may be slightly high because in our technique, we assume that all branches seen are associated with the source code. However, when filtering only application level branches, we will still see branches that we are not concerned with such as

branches from library and linking code.

The results of this experiment can be seen in Figures 5.7 and 5.8. By using the filter, samples contain more information that is useful in branch testing, allowing coverage rates to increase. Because fewer samples are necessary to achieve a high percentage of actual coverage, fewer interrupts occur, drastically reducing the time overhead. As can been seen for *sjeng*, the cost of gathering more than 63% of the branch data has an associated overhead of about 5% less than instrumentation. In this way, a filtering mechanism can substantially improve the applicability of the LBR for branch testing.

Multicore Sampling

Finally, we examine one way that multiple cores can be leveraged in conjunction with Event-Based Sampling. In this scheme, the initial sampling rate is divided by the number of available cores, 8 in our case. Each core then executes a copy of the program and monitors the LBR based on a shifted window. For example, with an initial polling period of 10,000 and 4 cores, the first core would monitor at 10,000, 20,000, and so on as normal. The second would take its first sample at 2,500 branches and proceed to sample every 10,000 branches after that. The third would be at 5,000 and the fourth at 7,5000. Using this approach, coverage precision similar to that achieved by the $\frac{\texttt{initial period}}{\texttt{\# of cores}}$ sampling rate can be achieved.

By sampling with a shifted rate across multiple cores, the percent of actual coverage is increased, on average, by 11.52% over single core monitoring, as can be seen in Figure 5.6. In the case of *sjeng*, the percent of actual branch coverage increases by nearly 21% in the 50 million branch sampling period. Unfortunately, the overhead of sampling also greatly increases when monitoring all cores at once. For example, although multicore monitoring every 50 million branches of *sjeng* obtains similar coverage to sampling every 6.25 million branches, as expected, the time overhead is also similar or slightly larger.

5.5 Conclusion

The work in this chapter represents the first look at exploiting hardware mechanisms for executing test cases while evaluating branch coverage. Experiment analysis shows that while the BTS is prohibitively expensive for use in branch testing, the LBR shows much potential in enabling a low overhead but effective branch testing technique Its use is especially promising because of its extremely low memory overhead averaging 0.57%. As most monitoring tools using instrumentation have memory overheads averaging from 60 to 90%, the use of the LBR can be advantageous in memory constrained environments where code instrumentation is too costly. The time overhead can also be improved relative to using instrumentation when only an estimate of complete coverage information is needed, such as when estimating fault-finding ability for test selection.

Applying a filter to the LBR provides us with our most promising results regarding the potential of using the LBR for effective branch testing monitoring. Filtering greatly decreases the amount of samples that need to be gathered during the monitoring process and helps ensure the relevance of the samples taken for branch testing. Thus, both efficiency and effectiveness are improved when applying a filter.

Hardware performance monitoring interfaces that expose monitors to user space are relatively new. OProfile [64], PAPI [19], and Perfmon2 [31] were each released in the last eight years and have been under steady development. As new mechanisms are added to processors, support for them is added as appropriate use cases arise. The BTS has been incorporated into libpfm4 since October 2009, and a patch for the LBR was proposed in March 2010. The LBR patch has yet to be added to the main kernel tree. Since the initial LBR patch was proposed, LBR sampling randomization and filtering have been discussed. When user support is added for LBR filtering, the LBR will be able to be used as an extremely low overhead branch testing tool.

In the next chapter, we further explore how the LBR in its current state of support can be exploited for efficient and effective branch testing in a system that we call THeME:

Testing by Hardware Monitoring Events.

Chapter 6

THeME: Testing by Hardware Monitoring Events

Contents

2.1 Test Suite Design and Analysis 17

2.1.1 Evaluating Test Suite Quality 18

2.1.2 Measuring Test Suite Effectiveness 20

2.2 Related Work 22

2.2.1 Test Selection and Prioritization 22

2.2.2 Executing Test Cases Efficiently 25

2.2.3 Hardware Performance Monitoring and Sampling 26

The LBR has much potential to be used for efficient and effective branch coverage monitoring, as seen in Chapter 5. However, there are several modifications that can be made with regard to how LBR samples are accessed, recorded, and supplemented that will improve our existing technique for determining the quality of tests being executed.

In this chapter, we explore the potential of using hardware mechanisms and multicore technology in branch testing, and we thoroughly evaluate the tradeoffs of leveraging these technologies for branch monitoring. We first evaluate a pure hardware approach to branch testing. In this exploration, we investigate two ways of accessing and reading hardware mechanisms, namely using OS polling and OS interrupts. Analysis of our techniques demonstrates the benefits of leveraging hardware advances in terms of time overhead and code

growth. Additionally, we evaluate how performing branch testing using hardware sampling affects the completeness of coverage monitoring. Next, we analyze the effects of integrating hardware monitoring information with the compiler infrastructure, which improves the completeness of coverage monitoring through the use of standard analysis techniques. Finally, we explore how multiple cores can be used in conjunction with hardware monitoring to improve the time overhead of testing.

This chapter provides empirical evidence that hardware monitoring can be adapted for more efficient branch coverage analysis compared to using instrumentation. Although hardware mechanism sampling leads to lossy test coverage information, it provides a promising low-overhead alternative to program instrumentation and can be used along with compiler analyses to attain upwards of 90% of the actual code coverage information. Hardware monitoring also requires only minor or no alterations to the program under test, making hardware approaches ideal in memory constrained environments where testing generally cannot be performed without simulation. Our techniques also enable the testing of multithreaded and time-sensitive code.

6.1 Improvement Challenges

There are several challenges in using the LBR to help solve the test execution problem. The first challenge deals with when the LBR is accessed. The LBR monitors all branches including those executing during program setup, teardown, library calls, and exceptions. Thus, monitoring should be limited only to program sections with which we are concerned. A second challenge relates to how the LBR is accessed. We should not attempt to access the LBR prematurely. Rather, we should try to access the LBR only when new information is available.

Finally, a balance must be found between the number of times that we access and record from the LBR and the completeness of branch information attained. In our initial THeME system, we take a pure hardware approach to sampling. However, the completeness of

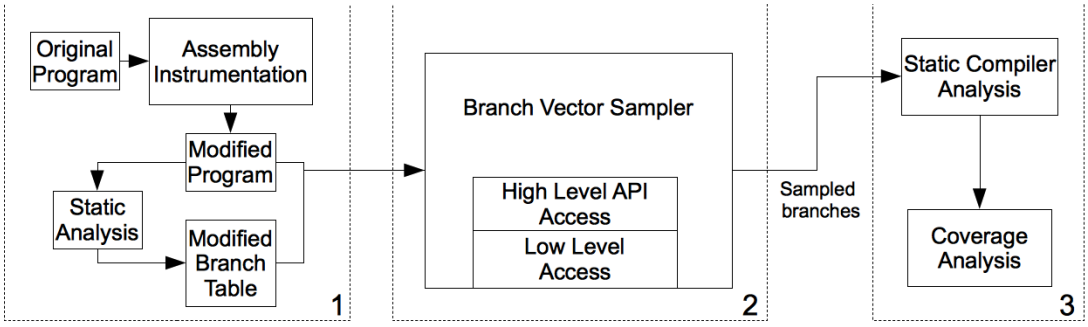


Figure 6.1: Infrastructure to adapt LBR monitoring to branch testing.

coverage monitoring can potentially be supplemented with standard analysis techniques and information from multiple cores.

6.2 Accessing and Sampling Branch Vectors

In order to evaluate the potential of using a purely hardware approach for structural testing, we first describe our branch testing infrastructure, which performs monitoring based on branch vector samples from the LBR. We then examine two different ways of accessing and reading branch vector data from the LBR. In other software tasks, such as path profiling and dynamic optimizations, larger sampling rates can be used to generate an estimation of program behavior as necessary. However, in software testing, smaller rates should be used to more thoroughly procure taken branch information. Thus, access speed is critical.

6.2.1 Implementation Details

An overview of our branch testing framework can be seen in Figure 6.1. Our infrastructure is executed on the system described in Chapter 5.4.1, and the same test applications are used.

Prior to beginning program execution with test inputs, we again give the branch-based mechanism the potential to observe the fall-through path by inserting harmless unconditional branches along every fall-through edge in the binary, as pictured in Figure 6.1(1).

	test			
Benchmark	Branch Cov.	Time (s)	Mod. Time (s)	Instr. Time (s)
bzip2	63.49%	16.5	16.9	18.6
h264ref	27.53%	43.8	43.8	47.7
libquantum	37.79%	0.155	0.16	0.165
mcf	73.70%	3.66	3.86	4.08
sjeng	46.29%	6.92	7.74	8.96
	ref			
Benchmark	Branch Cov.	Time (s)	Mod. Time (s)	Instr. Time (s)
bzip2	64.20%	1499	1514	1599
h264ref	35.72%	1753	1786	1890
libquantum	39.07%	1056	1178	1236
mcf	74.01%	529	539	575
sjeng	48.87%	1028	1162	1312

Table 6.1: SPEC 2006 benchmark time overhead information.

Our fall-through enabling modification tool generates programs that are on average only 0.5% larger and have 5% time overhead compared to native execution, as described in Chapter 5.4.2. The time overheads incurred by the modification tool for the *test* and *ref* SPEC inputs are shown in Table 6.1.

Once the program has been modified to enable fall-through branch monitoring, a simple static analysis is then used to identify the branch edges in the program's source code, as in traditional testing techniques. The branch edges are stored in a hash table along with information pertaining to the associated source code lines. This branch table is used as a checklist of branches with which we are concerned and is later used to calculate overall branch coverage.

Next, the test program is executed, as shown in Figure 6.1(2). We set LBR monitoring to begin when the test program enters its main method, and branch recording continues until the last instruction before the program ends. This prevents observation of the setup and teardown instructions executed as the program is loaded into and taken out of memory. Samples are taken based on the number of CPU Cycles observed during execution. When the sample rate of cycles is reached, the branches in the LBR are read and compared against the items in the branch table, and observed branches are marked as taken. Once the program under test has finished executing, the sampled branch coverage is calculated based on the number of source code-level branches observed divided by the total number of

source code-level branches in the program, as pictured in Figure 6.1(3).

The efficiency of our infrastructure is calculated based on the base run times of benchmark execution reported by the execution tool of the SPEC2006 benchmarks, `runspec`. All timing results are compared to the overheads observed from execution of full software-instrumented versions of the benchmarks. TestCocoon [106] was used to generate the instrumented benchmarks.

The effectiveness of our infrastructure is analyzed based on branch coverage.

6.2.2 User-level Branch Vector Access

There are a number of ways to access the branch vector data contained in the LBR, although most techniques in profiling, debugging, and other software tasks use some form of user-level performance monitoring API. Several available APIs include OProfile [64], PAPI [19], and Perfmon2 [31]. However, none of these yet support LBR reading. We use libpfm4 and its kernel-level interface, perfevents [31]. As in Chapter 5.4.1, we modified perfevents and libpfm to include LBR support.

6.2.3 Access via Polling

The simplest technique to access and read the LBR is through libpfm4's performance monitoring API and Linux's `poll` event. The test program is first spawned and executed using `ptrace`. Once the program has started successfully, LBR is enabled through a high level call to the operating system, as is the hardware counter that is to be used to trigger sampling. The monitoring program then repeatedly calls `poll`, which waits for the file descriptor associated with the performance counter to contain data that can be read, as shown below.

```
for(;;) {
        ret = poll(pollfds, 1, -1);
        if (ret < 0 && errno == EINTR)
                break;
        process_smpl_buf(file_descriptor);
}
```

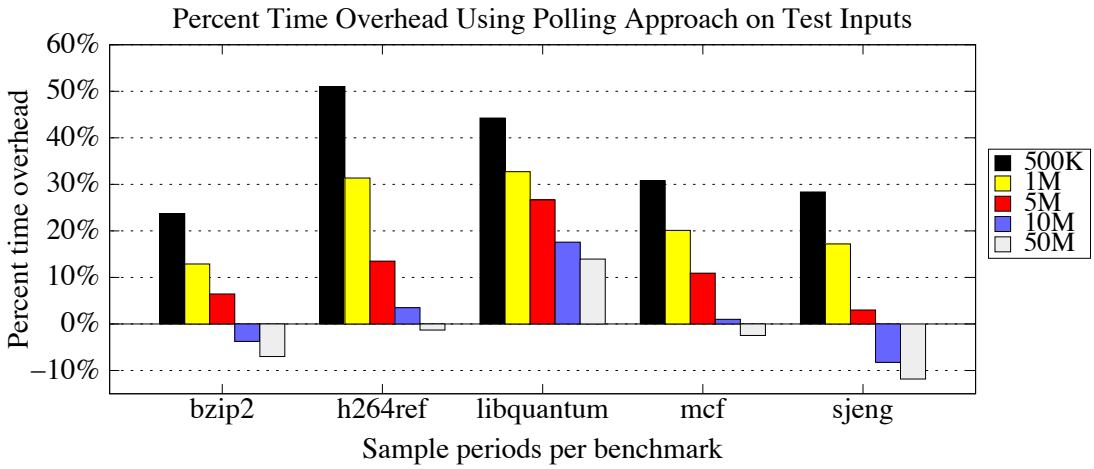


Figure 6.2: Time overhead for LBR sampling accessed using polling relative to full instrumentation.

Figure 6.2 shows the time overhead of branch testing when accessing the LBR using the polling approach relative to full software-level instrumentation. The results for running on the `test` inputs of the SPEC benchmarks are shown. While `poll` is an effective technique to report sets of LBR and performance counter data, repeatedly calling `poll` when no data is available causes unnecessary overhead. At sampling rates of 10 and 50 million, the polling approach improves time overhead slightly compared to the use of full instrumentation, performing with 12% less overhead than instrumentation in the case of `sjeng`. However, as sampling is performed more frequently, the cost due to repeatedly polling quickly rises. For example, sampling the LBR every 500K CPU cycles for `h264ref` results in 51% time overhead over instrumentation.

6.2.4 Interrupt Driven Access

In our second technique, we replace the repetitious call to `poll` with a lower level, more efficient hardware access approach. The hardware counters and LBR are enabled in the same way as described in Section 6.2.3. The `poll` calls are replaced by an I/O signal handler associated with our desired hardware mechanisms. The signal handler is immediately triggered upon the associated performance counter's overflow. After performing several checks, the

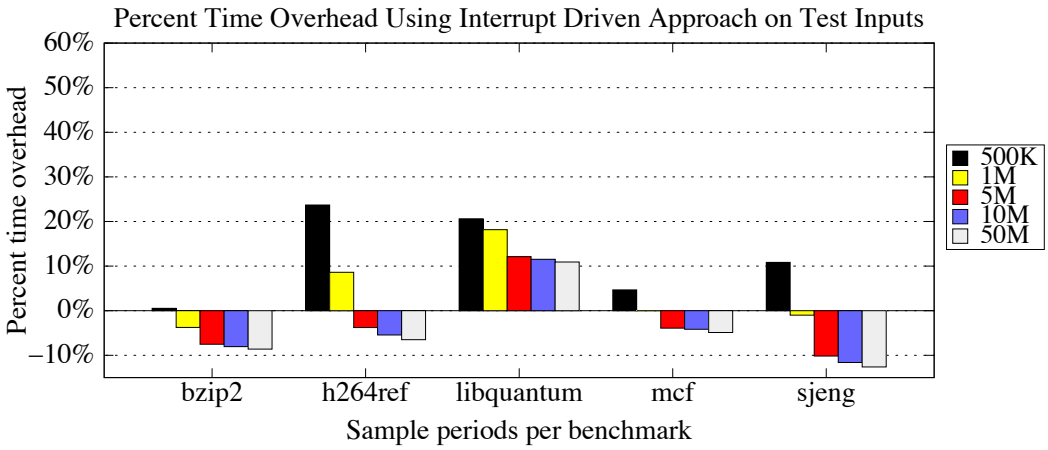


Figure 6.3: Time overhead for LBR sampling accessed using an interrupt-driven approach relative to full instrumentation on `test` inputs.

signal handler reads the LBR branch vector, and each branch is processed. The associated hardware counter then is reset and the program is resumed. By handling the performance counter notification and refreshing the counter directly from within the monitoring tool, we expect to significantly reduce the overheard associated with accessing and gathering data.

Figure 6.3 shows the time overhead of branch testing when accessing the LBR using the interrupt-driven approach for five sampling rates. The benchmarks were executed on the `test` inputs of the SPEC benchmarks, and the time overhead is compared to that of full software-level instrumentation. Using the interrupt-driven approach for access substantially improves the time overhead of gathering branch vectors compared to the polling approach. At sample rates of five, ten, and fifty million, the time overhead of branch testing is improved over instrumentation for all benchmarks other than *libquantum*. This is because *libquantum* only executes for 0.155 seconds, as seen in Table 6.1, and its percent time overhead is greatly impacted by any amount of noise. *Sjeng*'s time overhead, however, can be reduced by 13% compared to instrumentation.

To better understand the effects of sampling the LBR on time overhead and coverage, we next evaluate the time overhead and branch coverage measured when reading the LBR every 500 thousand, 1 million, 5 million, 10 million, and 50 million CPU cycles while executing

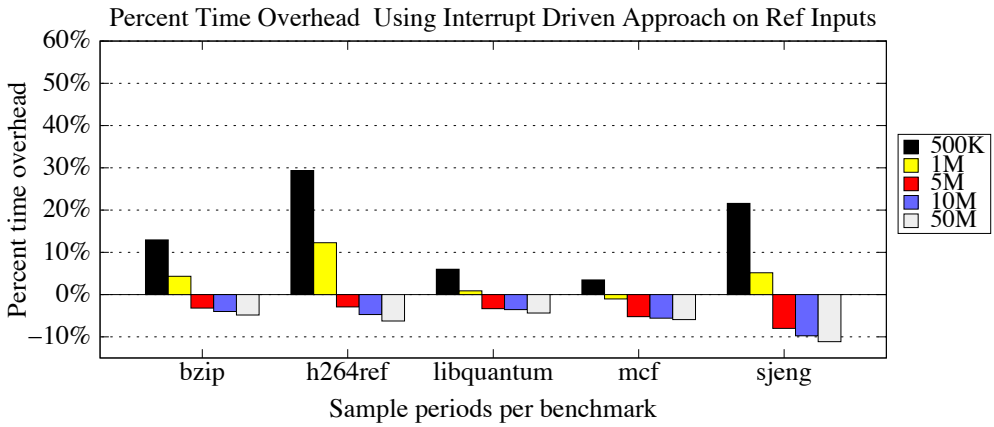


Figure 6.4: Time overhead for LBR sampling accessed using an interrupt-driven approach relative to full instrumentation on `ref` inputs.

the *ref* inputs of the SPEC 2006 benchmarks. Each benchmark executes an average of 19.55 minutes, as shown in Table 6.1. The time overhead of executing larger programs with LBR sampling increases when sampling at smaller rates such as 500 thousand. This is potentially due to the operating system becoming overloaded with interrupts at lower sampling rates. At higher rates (e.g. 5 million, 10 million, 50 million), the time overhead incurred shown in Figure 6.4 is consistent with the time overhead when executing on the *test* inputs.

On average, 76% of the actual coverage reported by instrumentation is observed when sampling the LBR every 500 thousand CPU cycles, as seen in Figure 6.5. *Sjeng* achieves 82.61% of the actual coverage, although the time overhead at that rate is 21.57% worse than instrumentation. However, at a sample rate of 50 million, *sjeng* still achieves 70.15% of the actual coverage while executing 11.13% faster than when monitoring using instrumentation. At a sample rate of 50 million, the average percent of actual coverage is reduced to 54%. When higher sampling rates are used, the time overhead of LBR monitoring is improved over instrumentation. However, higher sampling rates also correspond with lower effectiveness.

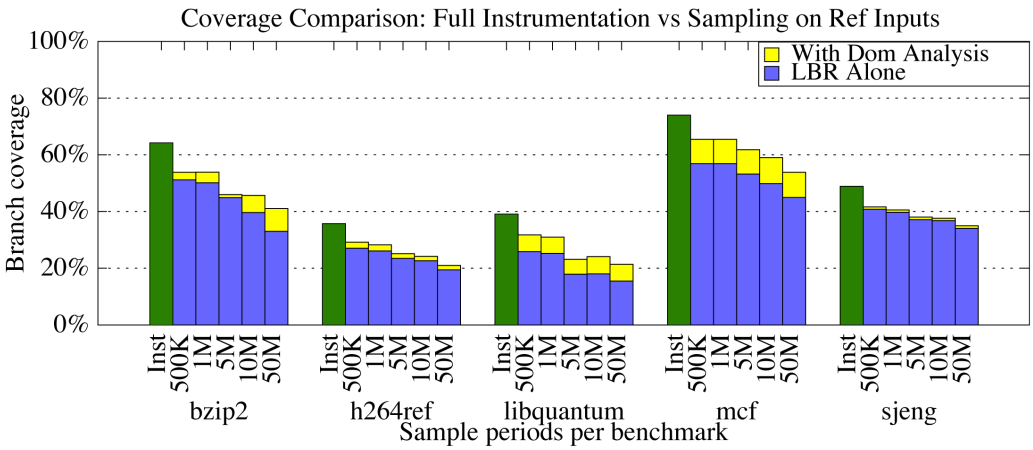


Figure 6.5: Coverage observed using LBR sampling via the interrupt-driven approach on `ref` inputs compared to instrumentation.

6.3 Improving Branch Coverage at High Sample Rates

In order to improve the branch coverage observed using a pure hardware approach to branch testing, monitored coverage details may be extended using offline compiler-based analyses. These analyses can be performed offline or on a separate core during program execution. We first associate the branches observed by the LBR with branches in the control flow graph representation of the program. Dominator and post-dominator analyses are then executed on the control flow graph to build a dominator tree.

Within a dominator tree, a basic block b dominates basic block c if every path from the entry of the control flow graph to basic block c contains basic block b. A basic block b post-dominates basic block c if every path from c to the exit of the CFG contains basic block b. For example, Figure 6.6 shows a control flow graph of a function in which the LBR has observed branch 5-7. Because basic blocks 5 and 7 were executed, blocks 1 and 2 must also have executed based on the dominator analysis. Blocks 8 and 11 also necessarily executed based on the post-dominator analysis.

Based on these two analyses, it is inferred that the conditional branches 1-2 and 2-5 must have executed, as well as the unconditional branch 7-8. Note that our branch testing technique only monitors conditional branches. However, when full branch vectors

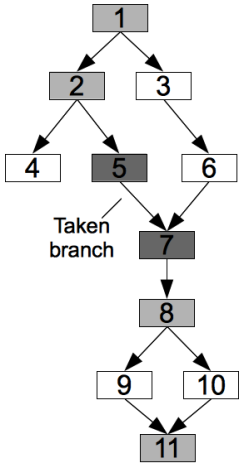


Figure 6.6: Dominator analyses based on an observed branch.

are observed, more branch vectors may be implied.

Our dominator analyses were executed using the LLVM compiler infrastructure [59]. By incorporating these two analyses, the percent of actual coverage observed was improved to an average of 83% across all benchmarks at a sample rate of 500 thousand, as shown in Figure 6.5. At a rate of 50 million CPU cycles, the average percent of actual coverage is improved to 62.32% from 54% without dominator analyses. *Mcf* achieves 90% of the actual test coverage with a sampling rate of 500 thousand and 72% with a sampling rate of 50 million. Thus, we find that supplementing LBR samples with information from simple static analyses that are already performed by the compiler can greatly improve coverage results.

6.4 Testing over Multiple Cores

Each core of a multicore processor contains its own hardware counters and mechanisms, which enables us to simply and efficiently monitor branch coverage in both sequential and multithreaded programs. We next observe the effect of monitoring test execution on multiple cores.

Our multicore experiments focus on the two of the five SPEC2006 benchmarks tested

in this research that include multiple inputs in the *ref* test set. Each input is executed on a separate core, and the coverage results were aggregated across cores as each test execution completed. The same sample rate was used on each core. The reference input set of *bzip2* includes six inputs, and *h264ref* includes three. Because our experiments are run on a quad core machine, we executed only the first four inputs to *bzip2*.

As shown in Figure 6.7, the time overhead of monitoring the execution of the first four inputs of *bzip2* using the LBR was 4% to 14% less than when using instrumentation. By removing instrumentation, the time overhead of executing test inputs on each core is improved, enabling greater time savings than when executing on a single core. As expected, the percent of actual coverage observed was the same as when executing on a single core, shown in Figure 6.5.

Unlike the overhead incurred by monitoring *bzip2*, the time overhead for *h264ref* using the LBR was greater than that of using instrumentation at sample rates of 500 thousand and one million. The timing results for *h264ref* are only slightly lower compared to sampling and executing on a single core. This is due to the fact that one of *h264ref*'s inputs executes for approximately 82% of the total execution time of the three inputs. Thus, the savings from executing the other two inputs on separate cores is not enough to substantially reduce the overall time overhead of monitoring *h264ref* using multiple cores versus a single core.

These experiments demonstrate that the time overhead of monitoring across multiple cores, relative to using instrumentation on multiple cores, incurs lower time overhead than when monitoring the LBR on a single core, relative to using instrumentation on a single core. Therefore, branch coverage analysis of multithreaded programs that execute on multiple cores will experience similar benefits to those of sequential or multithreaded programs executing on a single core. When the workload is evenly divided between multiple cores, we expect to observe time overhead results similar to those of *bzip2* in Figure 6.7.

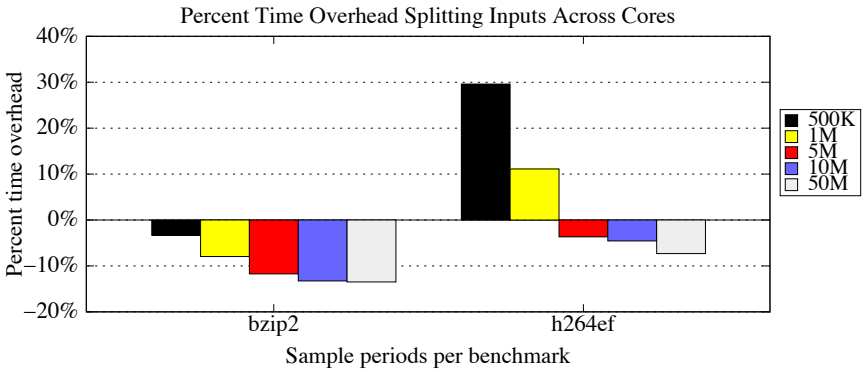


Figure 6.7: Time overhead for LBR sampling over multiple cores compared to using instrumentation on multiple cores.

6.5 Discussion

From the experimental results, we see that the LBR shows much potential in enabling a low overhead but effective branch testing technique for single and multithreaded programs. Used in conjunction with dominator analyses, LBR monitoring achieved up to 90% of the coverage observed using instrumentation with reduced time overhead and negligible memory overhead. In this section, we discuss several unique advantages that leveraging the LBR has over using instrumentation. We also describe software and hardware-level approaches that have the potential to further improve the efficiency and effectiveness of our techniques.

6.5.1 LBR Monitoring Benefits

The use of hardware mechanisms mitigates and removes the costs incurred by instrumentation, with additional benefits. The first advantage of leveraging hardware mechanisms rather than using instrumentation is with regard to the time overhead. Instrumentation must be inserted throughout a program at all points that are to be monitored, and it often remains in the program throughout execution. The cost of insertion and the time overhead incurred by repeatedly executing the instrumentation probes and payloads can be large. When monitoring using hardware mechanisms, however, a counter and mechanism need only be set up once during execution, and reading of the mechanism is inexpensive.

A second advantage of applying hardware mechanisms compared to using instrumentation is the lack of code growth. The code growth incurred by instrumentation is impacted by the size of the probe and payload and the frequency of insertion. Hardware mechanisms, however, require little or no program modification to perform monitoring.

The use of instrumentation is also inflexible in that only that which is instrumented can be monitored. To improve understanding of program execution, more instrumentation must be added, further increasing the time overhead and code growth of monitoring. Instrumentation traditionally must be added into a program's source code or binary. Hardware mechanisms, however, can be used to monitor multiple events at both the user and kernel levels. Thus, instead of analyzing only a section of program execution, hardware mechanisms can also report events that occur outside the source such as in library calls and external routines, painting a much fuller picture of program execution.

6.5.2 Improving Efficiency by Advancing Hardware

By modifying our methods of accessing the LBR branch information as discussed in Section 6.2, we greatly improved the efficiency of our branch monitoring techniques. However, there are a number of promising opportunities to advance the current usage model, practices, and implementation of hardware performance monitoring technology that could further improve our access schemes.

Elimination of OS Shepherding - In current systems, the kernel is required to shepherd all functions related to configuring, accessing, and reading hardware mechanisms. Requiring the operating system's involvement in all of these functions comes at a cost that is significantly higher than is necessary. At the lowest level, completion of these operations requires either reading or writing registers on the processor core, which is highly efficient by nature. However, when each operation is performed via the operating system, there are a number of added sources of overhead. These include an added system call to enter the privileged kernel mode, the saving of context by spilling user-level state to memory, and restoring this state when execution is returned to the user-level application.

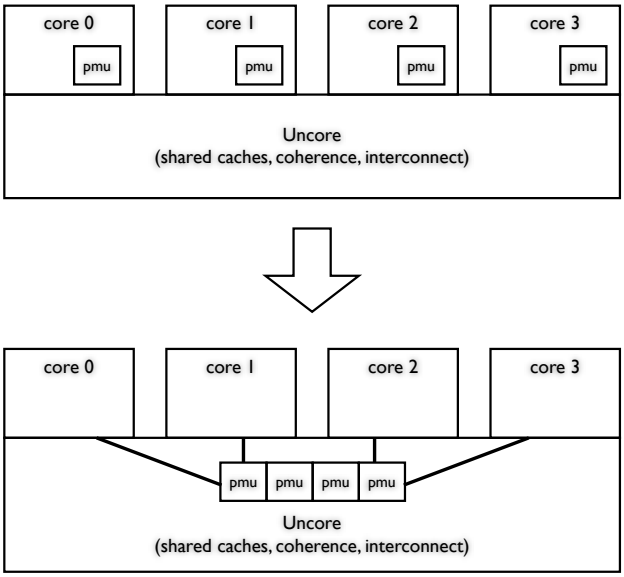


Figure 6.8: Moving private performance monitoring units to a global space to enable Satellite Monitoring.

For many of the traditional applications that use hardware monitoring, this kernel-level usage model is sufficient. Generally, hardware counters and mechanisms are only set up and torn down at the beginning and end of an application's execution. Often, the monitored counter is left to increment until the end of the application's execution, at which point, it is read. Thus, kernel involvement is required only twice. In other monitoring techniques, hardware mechanisms can be accessed and read infrequently during program execution [70, 71]. However, because test coverage analysis requires more frequent monitoring, requiring OS involvement on each sample severely effects the time overhead of analysis. Allowing counters to be accessed directly from user mode, would result in a significant reduction in the cost of accessing the hardware mechanisms [100].

There are two ways to achieve user control of hardware mechanisms. One requires hardware modification, while the other can be done using current hardware. The first approach is to allow the operating system to control the access permissions of hardware mechanisms directly by adding a simple register that can be used to specify execution modes that have direct access. While the overhead in the chip's die area to support this

added permissions mode would be negligible, hardware modification would be required. The second approach is to have the kernel not set the mode of the processor back to user mode when execution returns to the application of interest. This technique would only require modifying the OS implementation, but it would result in a security hole that can be exploited by malicious programs.

Modern processors include a set of bits that records the usage mode of the processor. If these bits are set to kernel mode, the registers associated with configuring and using hardware monitors are allowed to be read. Otherwise these registers are not accessible. To extend the permissions of the hardware monitors, a simple register can be used to specify the modes that can have direct access. The overhead in the chip's die area to support this added permissions mode would be negligible. This approach, however, would require hardware modifications. The second approach is to have the kernel not set the mode of the processor back to user mode when execution returns to the application of interest. This technique would only require modifying the operating system implementation. There is a disadvantage to this approach in that it would result in a security hole that can be exploited by malicious programs. However, if the user (tester/developer) is trusted, this kernel level access can be applied to only those programs that are being tested.

Satellite Monitoring - Another opportunity to improve hardware monitoring efficiency is to enable what we call *satellite monitoring*. Currently, hardware monitoring information can only be collected from the core hosting the application being monitored. This necessitates that the program be interrupted to collect the needed information. However, allowing hardware monitoring information to be accessed from any core would require minimal hardware modification. It would require moving each core's performance monitoring unit (PMU), which controls hardware monitoring ability, into the "uncore."

Figure 6.8 illustrates moving each private PMU to the global uncore area. Making the aggregated PMU universally accessible would require an added core id assigned to each global PMU and added bus lines from each core. This approach would allow the monitoring and analysis of the application from a core separate from those hosting the

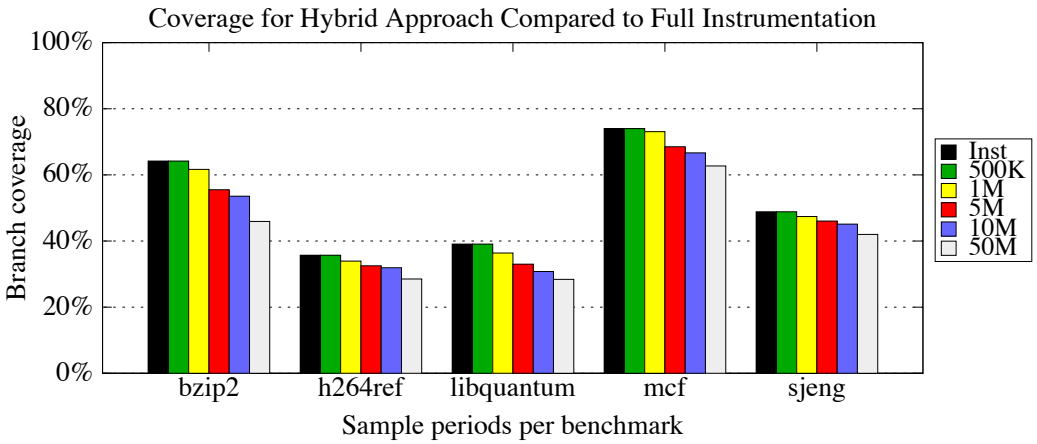


Figure 6.9: Percent of coverage observed when selectively instrumenting branches compared to instrumentation.

application's threads, allowing the application to be unperturbed throughout execution. Using this approach would allow us to combine the advantages of multicore with those of performance monitoring technology.

6.5.3 Improving Effectiveness Through Instrumentation

The effectiveness of our techniques could also potentially be improved by inserting a small amount of software-level instrumentation into the application. Samples from hardware mechanisms are much more likely to observe instructions that occur along frequently executed paths. Instructions that are only hit occasionally, however, are unlikely to be seen.

Ideally instrumentation would be added only to branches that are unlikely to be observed in LBR samples, and it would be inserted dynamically or prior to executing the test suite. However, identifying these locations is challenging. Without prior knowledge of program execution or profile information of the application, it is unclear where instrumentation should be inserted. If instrumentation is added unnecessarily, the overhead improvements from leveraging the LBR will be reduced. On the other hand, a conservative approach may have little impact on improving efficiency.

Figure 6.9 shows the coverage results of selectively instrumenting the benchmark appli-

cations when prior knowledge of execution, as reported by the LBR, is available. For each benchmark, instrumentation is added along any branch edge that was not observed by the LBR with a sampling rate of 500 thousand. Then the application is executed a second time to calculate the coverage obtained from both hardware and instrumentation. As is shown in Figure 6.9, at a sample rate of 500 thousand, nearly complete coverage information is observed in all cases. At a sample rate of 50 million CPU cycles, the percent of actual coverage reported by a pure instrumentation approach is reduced to an average of 80%.

These results are promising in terms of effectiveness. However, for a hybrid approach to be applicable for structural testing, we should assume that there is no prior analysis information available for the application and that the test suite only needs to be executed once to calculate coverage.

6.6 Conclusion

The work in this chapter demonstrates that hardware mechanisms and multicore technology can be adapted for use in efficient and effective branch coverage analysis during test execution. We developed a runtime system that performs branch coverage analysis by monitoring hardware mechanisms on single and multiple cores. Monitoring program execution using hardware mechanisms was up to 11.13% faster in our tests compared to using instrumentation, but it does not provide complete coverage information. To improve coverage, we additionally perform a compiler analysis to extend the amount of coverage derived from each sample. The results show up to 90% of the actual code coverage can be determined with less time overhead and negligible code growth compared to using instrumentation.

Because these hardware approaches require only minor or no alterations to the program under test and incur low time overhead, they are ideal in resource constrained environments where testing generally cannot be performed without emulation. For this reason, they can also be applied to enable the testing of time-sensitive or multithreaded code.

In the next chapter, we explain how the THeME system can be extended to additionally

monitor for statement coverage using simpler hardware mechanisms that are more representative of hardware mechanisms available on common devices, including resource-constrained devices used in mobile computing.

Chapter 7

Executing Test Cases for Statement Monitoring

Contents

3.1 Time-Aware Selection 30

3.2 Knapsack Solvers as Selectors 30

3.3 Experiment Goals and Design 34

3.3.1 Case Studies 35

3.3.2 Evaluation Metrics 36

3.4 Experiments and Results 40

3.4.1 Selection Effectiveness. 40

3.4.2 Selection Efficiency. 42

3.5 Conclusions 44

Similar to branch coverage, statement coverage is also frequently used to measure the quality of a test suite. Statement coverage measures the percentage of executed statements to the total number of statements in the application under test [124]. A high level of statement coverage is correlated with fault-finding capability, although it is recognized that statement coverage alone may not be a strong indicator of software quality. This is because statement coverage produces very different results depending on how the source code is formatted [106]. For example, in the code shown below, statement coverage would be 33%.

```
1 int main()
2 {
3 HIT   if (true) return 1;
```

```
4 MIS   foo();
5 MIS   return 0;
6 }
```

However, if the code is reformatted, as shown below, statement coverage would be 66%.

```
1 int main()
2 {
3 HIT   if (true)
4 HIT     return 1;
5 MIS   foo(); return 0;
6 }
```

Despite these disadvantages, statement coverage is widely used in industry as a criterion for test quality [18, 25, 55, 92, 108]. Different standards require achieving high levels of statement coverage. For example, avionics industry standard `DO-254` demands that close to 100% statement coverage be achieved. Avionics industry standard `DO-178B` and automotive industry standard `IEC 61508` detail similar requirements.

In this chapter, we extend THeME to execute tests while monitoring statement coverage. We make the additional requirement that the source code cannot be modified in any way prior to execution. Therefore, we present a technique that uses only hardware mechanisms to monitor execution while requiring no code growth, recompilation, or compiler analysis tools.

The experimental results show that up to 79% of the statement coverage reported by instrumentation can be reported with lesser time overhead than instrumentation. Additionally, because THeME does not require modifications to the program under test, there is no code growth to the program, unlike in instrumentation.

Another significant advantage of our tool extension is that we require only common hardware mechanisms to be available on the device. More advanced mechanisms such as the LBR and BTS are available only on a small range of processor types, and the support for these mechanisms is lacking at the kernel and user tool levels. Instead, we take advantage of the `CPU_CYCLES` hardware counter, which is available on every tablet, smartphone, and

commodity computer that we have found. At the end of this chapter, we discuss the challenges and advantages of using our THeME system on such devices.

7.1 Challenges of Statement Monitoring

The first challenge of this work is in selecting a hardware mechanism for use in sampling. Again, there are a wide-range of hardware mechanisms to choose from, and many of these are supported for sampling use at the user and kernel levels. We want to select a mechanism that is widely available on many different types of devices. When using these simpler mechanisms, the time overhead of use is only dependent on the number of samples taken; each mechanism is accessed and information is recorded in the same way for all.

Therefore, as in branch testing, we again have the challenge of balancing the efficiency and effectiveness of our hardware monitoring tool. When using software-level instrumentation for statement coverage, probes and payloads are added for counting each time individual lines are executed during the program. Additionally, instrumentation code is inserted for each branch of the program, where branch instrumentation records how frequently different paths are taken through if statements and other conditionals. When sampling instructions using hardware mechanisms, any statement that is executed may be observed, although recording can be tied to particular processes if desired.

A third challenge comes from our requirement that the source code must not be modified. This requirement is advantageous because execution information can be determined for any executing program without the need of recompilation. By not modifying the code though, THeME loses its capability of starting and stoping monitoring when code that does not correspond to our program's source code, such as setup and teardown code, is executing. This will result in higher time overheads when executing the tests. However, current testing tools [44,69,76] can only achieve monitoring of unmodified code through the use of dynamic instrumentation, which is often significantly more expensive in terms of time and memory than static instrumentation.

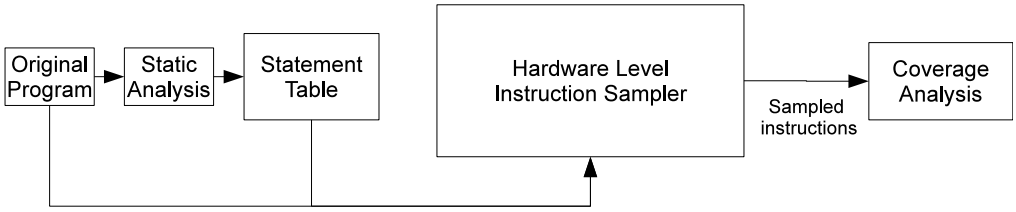


Figure 7.1: Infrastructure to adapt CPU_CYCLE monitoring to statement testing.

7.2 Hardware Monitoring for Statement Testing

To achieve the goal of exploiting hardware mechanisms for statement coverage, we first examine two common hardware mechanisms that can be used for sampling. After selecting a mechanism, we then experimentally evaluate the time and code growth overheads that sampling incurs and analyze the level of statement coverage that can be observed during sampling. Finally, we demonstrate additional information that can be obtained while monitoring test execution and discuss its potential usage within future work.

THeME is executed on the system described in Chapter 5.4.1, and the same test applications are used. The infrastructure of THeME is simplified for our statement coverage technique, as shown in Figure 7.1. Because hardware mechanisms are being used to observe statements rather than branches, no program modification is necessary. The program's binary is first analyzed to record all statements that may be executed. Debugging information is used to generate a table of statements, relating back to the source code. Then the table and program are passed to the hardware-level monitor for program execution. Upon program completion, the updated table is analyzed to calculate the overall statement coverage of our test inputs. For this chapter, we removed the static compiler analysis component present in Chapter 6. This was done for two reasons. First, the component removal allows us to more accurately observe what level of statement coverage hardware monitoring alone can discover. Second, in most resource-constrained devices, no compiler or build tools are available on the system. Thus, this revision to our system more precisely matches the statement coverage levels that could be observed on such devices.

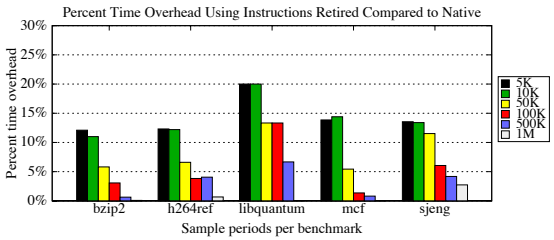


Figure 7.2: Time overhead for Instructions Retired compared to native on `test` inputs.

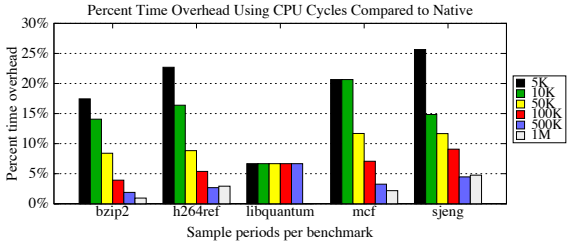


Figure 7.3: Time overhead for CPU Cycles compared to native on `test` inputs.

The efficiency of our infrastructure is calculated based on the base run times of benchmark execution reported by the execution tool of the SPEC2006 benchmarks, `runspec`. All timing results are compared to the overheads observed from execution of full software-instrumented versions of the benchmarks. In this chapter, gcov [5] was used to generate the instrumented benchmarks.

The effectiveness of our infrastructure is analyzed based on statement coverage.

7.2.1 Choosing a Mechanism

The time overhead and code coverage that is observed during sampling is heavily dependent on the hardware mechanism that is selected for use. In branch testing, the LBR was initially selected because it collects and reports partial paths of branches during execution. For statement coverage, the Instructions Retired monitor is intuitively the most appealing as the mechanism samples single instructions as they retire. However, CPU Cycles is another option and will possibly produce higher coverage reports than Instructions Retired at the same sampling rates.

To observe the impact of our selection on time and statement coverage, we first execute the benchmarks with *test* inputs over six different sampling rates while monitoring with the Instructions Retired and CPU Cycles. The time overhead of execution compared to native execution time is shown in Figure 7.2 for Instructions Retired and Figure 7.3 for CPU Cycles. Overall, the time overhead of using the Instructions Retired mechanism is less than that when using CPU Cycles. The `libquantum` benchmark is an exception, but this

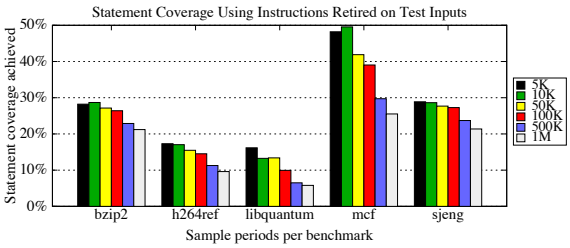


Figure 7.4: Statement Coverage using Instructions Retired on `test` inputs.

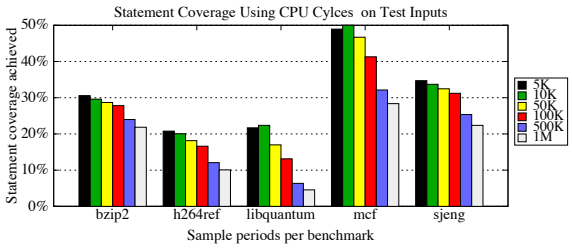


Figure 7.5: Statement Coverage using CPU Cycles on `test` inputs.

is because the execution time of `libquantum` is only 0.15s. Thus, even a 0.01s increase in reported time appears significant.

Although Instructions Retired produces less overhead than CPU Cycles on average, the effect can be seen in terms of statement coverage. Figures 7.4 and 7.5 show the coverages achieved when using both hardware mechanisms. The use of CPU Cycles results in a higher level of statement coverage than the use of Instructions Retired across all benchmarks and sampling periods because there are more samples taken during the life of the program. Although this results in a slightly higher time overhead, coverage improves by 14% on average across all benchmarks and sampling periods.

Notice that when sampling using CPU Cycles or Instructions Retired, much smaller sampling rates can be used than when sampling the LBR without greatly increasing the time overhead. This is because CPU Cycles and Instructions Retired are more primitive hardware mechanisms. When the LBR is used, there is extra cost associated with filling the LBR and reading a full branch vector. With CPU Cycles or Instructions Retired, however, overhead is only incurred by the interrupt and single sample reading.

7.2.2 Statement Coverage Comparisons

Because the use of CPU Cycles results in a higher level of coverage, the next set of experiments shows the time overhead, coverage, and code growth compared to that of instrumentation when executing the *ref* inputs of SPEC with CPU Cycles.

Figure 7.6 shows the time overhead incurred by THeME compared to the time overhead

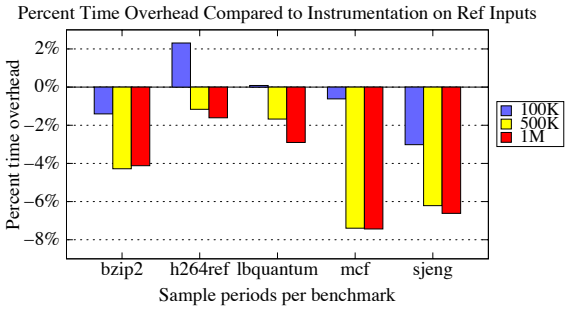


Figure 7.6: Time overhead relative to full instrumentation on *ref* inputs

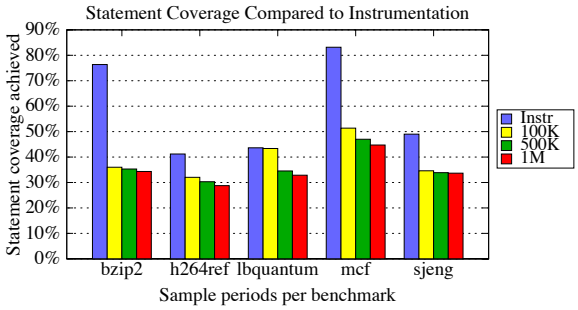


Figure 7.7: Statement coverage using CPU Cycles compared to Instrumentation on *ref*

Benchmark	Native Size (kB)	Instrumentation (kB)	CPU Cycles (kB)
bzip2	260	359.30	260
h264ref	2892	3549.88	2892
libquantum	208	255.59	208
mcf	128	138.93	128
sjeng	592	825.09	592

Table 7.1: Code Growth of Instrumentation vs CPU Cycles

of instrumentation. At a sampling rate of 100 thousand, there is a 0.6% improvement in time overhead on average. At 500 thousand, the overhead improves by 4.3%, and at 1 million cpu cycles, the time overhead improves by 4.5% on average.

In Figure 7.7, the statement coverage achieved by THeME is shown and compared to instrumentation. On average, 66% of the coverage reported by instrumentation is observed using CPU Cycles across the three sampling periods. At a sampling rate of 100 thousand, 71% of instrumentation's coverage is observed. At 500 thousand, 64% is observed, and at 1 million, 62% is observed. The execution of `libquantum` and `h264ref` report the highest coverage compared to instrumentation when sampling at 100 thousand cpu cycles with 77% and 99% respectively. This is a significant improvement over the coverage observed when sampling using the LBR for branch testing because in branch testing, we were observing single jumps in the program's binary. In statement coverage, many instructions in the binary may be associated with a single line of source code, which provides a greater opportunity for observing the execution of the source line.

An advantage of performing statement coverage using simple hardware mechanisms

comes from the fact that no code modification is necessary to enable our technique. Thus, THeME incurs no code growth. The code growth produced by instrumentation, however, is significant, as shown in Table 7.1. On average, statement level instrumentation using gcov produced binaries that were 40% larger than the original program. These size increases are smaller than those reported in the literature for coverage testing tools [76, 95, 106], meaning that on average, our system would be even more advantageous for use on memory constrained devices.

7.3 Discussion

The results show that up to 79% of the statement coverage reported using instrumentation can be observed using THeME with a reduced time overhead and no code growth. 79% is achieved when executing the `libquantum` benchmark with a sampling rate of 500 thousand. These results are promising, although the coverage and time overhead could be further improved.

The statement coverage produced by this tool extension would additionally benefit from a dominator analysis, as was described in Chapter 6.3. However, for the purposes of observing the capabilities of a pure hardware approach to statement coverage testing without the need for compiler based tools or special libraries, this portion of THeME was omitted. In environments in which we expect THeME to be particularly useful, such tools would not be available.

The time overhead could be improved by limiting sampling only to portions of program execution with which we are concerned. In addition to setup and teardown instructions, the samples observed by THeME include calls to libraries such as `libc-2.11.1.so` and a number of kernel functions that execute during program execution. For example, when monitoring the execution of *mcf* every ten thousand cpu cycles, approximately 29 million samples are taken. Of these samples, 0.04% samples are associated with kernel calls and 0.31% samples are from `libc-2.11.1.so`. By modifying our sampling techniques to not

include this extra data, the time overhead would be reduced.

7.4 THeME for Tablets and Smartphones

The execution and evaluation of test cases is ideally performed within the environment in which the final application will execute. When developing applications for memory constrained devices such as modern tablets and smartphones, however, tests are generally only executed using an emulator. Emulators tend to be much slower than the device itself, and they cannot accurately copy all conditions under which applications may run.

The process of executing test cases on the device itself is difficult. In the past year, attempts have been made at automating the testing process [32]. However, no research has yet been done regarding test case evaluation during execution on the device. Static instrumentation tools could potentially be ported to execute on the devices, but the memory overhead and code growth that they incur will likely stress the system and reduce the ability to fully and accurately execute tests. For example, tests such as those described in [101], would be severely limited if test execution itself increases memory overhead. Dynamic instrumentation tools would likely suffer the same issues. Additionally, because dynamic instrumentation techniques generate instrumented code during execution, they cannot simply be ported over for new systems.

The THeME system, however, provides an efficient and effective solution to the test execution process and can easily be ported to other devices provided that i) the devices have accessible hardware mechanisms, ii) there exists kernel support for accessing and reading the mechanisms, and iii) there exists a kernel interface that will properly interact between the access tool and the hardware. Most modern smartphone and tablet devices meet the first two requirements. For example, the iPhone 3GS, Nokia n900, Samsung Galaxy Nexus, iPad2, Motorola XOOM, and the Amazon Kindle Fire all use ARM Cortex-A8 or A9 processors. The Cortex-A8 an A9 have more than fifty hardware counters that can be utilized, and they are accessible at the kernel and user levels through the *perf* and *Oprofile* tools.

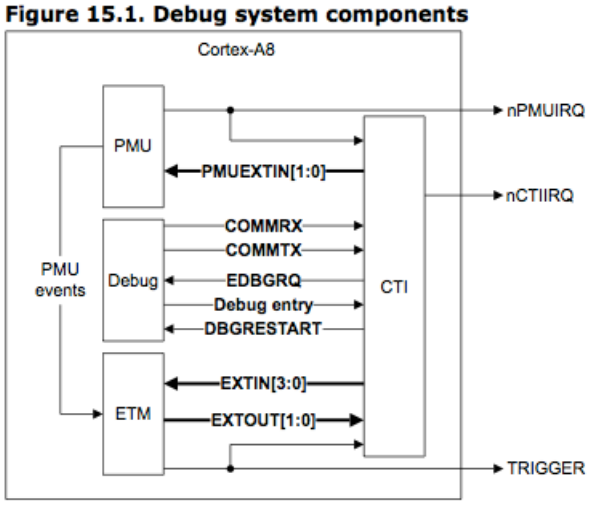


Figure 7.8: The Cross Trigger Interface [12]

The hardware mechanisms on the Cortex-A9 processor, which is used in most multi-core mobile devices, are unique in that they also support the tracking of code in Java applications. In addition to the typical mechanisms that track cpu cycles, branches retired, data read/writes, etc..., the A9's mechanisms also can monitor decoded Java bytecode execution and Jazelle backward branch execution [31,64]. Jazelle Direct Bytecode eXecution allows processors to execute Java bytecode in hardware as a third execution state. The most prominent use of Jazelle is by manufacturers of mobile phones to increase the execution speed of Java Micro Edition games and applications. A Jazelle-aware Java Virtual Machine (JVM) will attempt to run Java bytecodes in hardware, while returning to the software for more complicated, or lesser-used bytecode operations. ARM claims that approximately 95% of bytecode in typical program usage ends up being directly processed in the hardware [11]. Therefore, these two additional mechanisms potentially are very useful for monitoring the behavior of Java applications.

Although hardware mechanisms exist on processors for mobile devices and although recent kernels provide tool support for accessing and recording information from these counters, the kernel interface between the two is lacking. In order to take advantage of

hardware mechanisms, the access tool, kernel, and hardware must be able interact appropriately. As an example, many mobile devices are designed with the TI OMAP4 system on a chip (SoC), which is a high-performance applications processor. Devices that use an OMAP4 SoC include the Amazon Kindle Fire, Barnes and Noble Nook Tablet, Blackberry Playbook, and Motorola Droid RAZR. On the SoC, a cross trigger interface (CTI) enables the access of hardware mechanisms, which are controlled by the Performance Monitoring Interface (PMU), as shown in Figure 7.8. When a hardware mechanism is ready to be read, the CTI generates an interrupt. The CTI then sends the interrupt to a platform specific interrupt controller, which then routes the interrupt to a generic interrupt controller.

Many devices that operate with the TI OMAP4 SoC use some version of the Android operating system. However, all current versions of the Android operating system are built on Linux 2.6.35, which did not include support for the cross trigger interface or platform specific interrupt controller. While the CTI can be integrated into the kernel as a component, the platform specific interrupt controller is tied into fundamental linux kernel timing tasks and cannot be added in easily. Therefore, until the underlying kernel within the Android operating system is updated, hardware mechanism usage remains unsupported.

However, it is unlikely that the divergence between the Android operating system and the Linux kernel will continue, particularly as Android continues to grow in popularity. In the past, attempts at merging key Android changes with Linux have failed or ended up in staging only to be abandoned unmaintained. Yet, towards the end of last year, the Linux Foundation, Linaro, and others initiated a new effort to mainline Android changes into the Linux kernel [57]. These changes are becoming evident in Linux 3.4, the first version of which was released on March 31, 2012 [109].

With operating system support, THeME can easily be extended to support test execution using any of the available hardware mechanisms, and the system can be recompiled simply to be ported to modern mobile devices. Thus, THeME has the potential to provide an efficient and effective solution for test execution and monitoring on such resource-constrained devices.

Chapter 8

Merits and Future Work

Contents

4.1 Genetic Algorithms and the Test Selection Challenge 46

4.1.1 Designing a Genetic Algorithm 46

4.1.2 Genetic Algorithm Challenges 48

4.1.3 The Test Selection Challenge 49

4.2 Time-Constrained Selection 49

4.2.1 Overview 50

4.2.2 A Genetic Algorithm for Time-Aware Test Selection 50

4.2.3 Test Selection in Action 57

4.3 Empirical Evaluation 59

4.3.1 Experimental Design 60

4.3.2 Experiments and Results 63

4.4 Conclusions 75

This thesis develops and evaluates techniques for efficient and effective execution of test cases in resource-constrained environments. The first set of approaches selects test cases from large test suites through the use of knapsack approximation algorithms to optimize likely fault-finding capability within given resource constraints. The second set of techniques provides methods for evaluating test case quality during test execution while maintaining

a high level efficiency. Our experimental evaluation reveals that these approaches are ideal for executing test cases in environments where time, power, or memory are limited.

8.1 Contributions and Merits

There are two significant contributions of this dissertation. The first contribution is the demonstration that higher levels of coverage and fault detection are obtained when constraints are explicitly considered during the test selection process (see Chapters 3 and 4). We developed and evaluated eight techniques that can be used to select test cases specifically for execution within resource constrained environments. The first seven techniques use 0/1 Knapsack solvers to best select and prioritize test suites without taking code coverage overlap into account [9]. In the eighth technique, which uses a genetic algorithm to perform selection, code coverage overlap is additionally considered when evaluating the quality of potential test selections [115, 116].

Our results indicated that a trade-off must be made between the efficiency of the selection algorithm and final code coverage of the test selection. We additionally learned that the design of the test suite is of great importance. If test cases have been carefully designed and there is little overlap between the tests, a more efficient, non-coverage-aware test selector can be used with favorable results, comparable to those of an overlap-aware solver. However, if there is a large amount of overlap between test cases, the added expense of an overlap-aware solver would be worthwhile. While more sophisticated solvers such as dynamic programming, generalized tabular, and core are likely to obtain higher utility than simple solvers such as greedy by value and greedy by ratio, neither group can make any guarantee regarding final cumulative coverage of the result. Thus, if correctness of the program is of highest importance, the selection process time would be better spent using a solver that takes test case overlapping coverage into account in addition to the constraints that will be present during test execution.

The second major contribution is our exploration of the potential of exploiting hard-

ware mechanisms for use in efficient test quality analysis. Over the course of our work, we thoroughly evaluated the success of using three different hardware mechanisms for branch coverage and statement coverage monitoring (see Chapters 5–7). Our techniques allow for branch and statement-level monitoring using hardware mechanisms that incur lower overhead than traditional instrumentation approaches [100]. The techniques are additionally significant because they incur either no code growth or negligible code growth, depending on the structure being monitored. This makes our system ideal for execution in resource constrained environments.

Our empirical evaluations demonstrated that hardware mechanisms can be adapted for use in efficient and effective branch coverage analysis. We also conducted analyses revealing the benefits of testing using hardware mechanisms on multiple cores and a demonstration of how the compiler infrastructure can be used along with hardware mechanism monitoring for improved test coverage.

Applying our approaches, we developed a runtime system with static components that we call THeME: Testing by Hardware Monitoring Events [117]. THeME incorporates each of the techniques discussed in Chapters 5–7. The result is an efficient and effective tool for test execution and evaluation. The system is extensible in that other hardware mechanisms can easily be substituted in for those used. The system is also portable in that it does not rely on special libraries or tools. The hardware mechanisms used are available on commodity machines, tablets, and mobile devices, making THeME applicable for use on such devices, given kernel support.

8.2 Future Work

There are several future directions of research regarding selecting and executing test suites that we would like to pursue. These ideas for future work fall under the challenge of software testing and debugging within resource-constrained environments.

8.2.1 Selecting Test Cases Based on Estimations

The selection techniques that are presented in this dissertation rely on 1) accurate measurements regarding resource usage and 2) efficient techniques for evaluating test cases. The use of hardware mechanisms in test case execution reduces the resource-usage required of each test case and provides an estimate of coverage that can be used to estimate the fault-finding ability. By taking advantage of hardware mechanisms when executing test cases, we can potentially reduce the overhead of selection as well. The challenge is to be able to select based on the coverage estimations that samples from hardware mechanisms provide.

8.2.2 Combining Hardware Sampling and Limited Instrumentation

Sampling hardware mechanisms alone will produce incomplete coverage information, even when very small sample rates are used. Small sampling rates will also generate a much higher time overhead, possibly negating the advantages of harnessing hardware. Although software-level instrumentation is an expensive form of monitoring, its cost can be reduced by placing instrumentation only along infrequently executed paths. Frequently executed events are also more likely to be observed in samples from hardware mechanisms. The sampling of hardware performance monitors can be supplemented with software-level instrumentation to improve the efficiency and effectiveness of software testing.

The main challenge of supplementing hardware monitoring with software-based instrumentation is that it is unclear at what program points instrumentation should be added. To achieve complete coverage information, instrumentation can be added at events unobserved by hardware monitoring. In a two step execution process, this is simple, and execution of a partially instrumented program incurs little overhead. However, performing two separate runs is not suitable in a testing environment.

8.2.3 Execution for Evaluation of Other Test Metrics

Branch and statement coverage are simple and inexpensive to calculate from taken branch information, but other test metrics may be more effective for both sequential and multi-

threaded programs.

Data-Flow Analysis

One such metric is definition-use association (DUAs) coverage. Data-flow coverage metrics generally are more expensive to compute than branch or statement metrics, but they are often more effective in terms of fault finding ability [35].

One challenge in performing data-flow monitoring is due to the fact that data-flow testing examines the lifetime of data variables. Thus, partial path information must be tracked in addition to structural coverage. When performing branch testing, exact observed branch ordering is unnecessary. However, for data-flow analysis, the instructions must be reported precisely. For example, when calculating def-use association coverage, there may be more than one coverage order for the nodes that constitute a *du* pair, including *kill* nodes. This occurs when two or more *du* nodes are mutually reachable or enclosed in a common cycle. No hardware mechanisms exist that will report a vector of memory instructions, like the LBR for branches, so our monitoring and analysis techniques must sample more thoroughly and extrapolate executed instructions based on those observed. Also, when monitoring taken data, execution ordering information must also be recorded.

Metrics for Multithreaded Programs

Many coverage metrics have been proposed for multithreaded programs, but there is little known about their effectiveness in determining fault-finding ability [68]. The use of hardware mechanisms for multithreaded program execution is interesting because each core on a multicore processor has its own set of hardware mechanisms. However, future techniques must account for inter-thread behavior when analyzing multithreaded programs. In the case of branch coverage, our results indicate that tests for multithreaded applications can easily be evaluated. For more complicated coverage metrics such as those related to data-flow, information from multiple cores can lead to a substantially more challenging problem.

8.2.4 Hardware Mechanisms and VM Environments

While hardware mechanisms work well for monitoring execution as it executes on the processor, it is unclear how the information obtained can be applied to programs executing within a Virtual Machine (VM). Programs written in languages such as Java that execute in virtual machines are common, and they are particularly growing in popularity due to the rise of mobile computing. The challenge is to be able to determine correlations between the instructions that are observed by the hardware and the instructions that are executing on the VM.

8.2.5 Fault Localization

Finally, hardware mechanisms such as the LBR are likely to be very useful in debugging and fault localization. After taking an LBR sample during program execution, we have a concrete path of branch execution leading up to that point. Additional samples earlier in program execution can provide additional branch path information. At the point of a fault, a trigger can be inserted to determine the most recently executed branches taken that lead up to the fault, and more triggers can be added to sample earlier in program execution. Determining locations for additional triggers is challenging, but we believe that with a proper trigger-insertion scheme, paths of execution leading to faults can be observed.

Bibliography

[1] Android developers: Using hardware devices. http://developer.android.com/guide/developing/device.html.

[2] Cobertura. http://cobertura.sourceforge.net/.

[3] Cyanogenmod nightly builds. http://www.cyanogenmod.com/blog/cm7-nightly-builds.

[4] Firefox nightly builds. http://nightly.mozilla.org/.

[5] gcov. http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html.

[6] mozilla - developer central. http://www.mozilla.org/developer/.

[7] Nightly Build - XBMC. http://wiki.xbmc.org/index.php?title=Nightly_build.

[8] VideoLAN, VLC media player continuous nightly builds. http://nightlies.videolan.org/.

[9] S. Alspaugh, K. R. Walcott, M. Belanich, G. M. Kapfhammer, and M. L. Soffa. Efficient time-aware prioritization with knapsack solvers. In *WEASELTech07: Proceedings of the ASE Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, Atlanta, Georgia, November 2007.

[10] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Not.*, 32(5):85–96, 1997.

[11] ARM. ARM Jazelle technology. http://www.arm.com/products/processors/technologies/jazelle.php.

[12] ARM. *Technical Reference Manual- Revision: r1p1*. ARM Limited, 2006.

[13] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. *SIGPLAN Not.*, 36(5):168–179, 2001.

[14] Atlassian. Clover: Java and groovy code coverage. http://www.cenqua.com/clover/.

[15] R. Azimi, M. Stumm, and R. W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 101–110, New York, NY, USA, 2005. ACM.

[16] D. Binkley. Semantics guided regression test cost reduction. *IEEE Trans. Softw. Eng.*, 23(8):498–516, Aug. 1997.

[17] R. Bodík, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. *SIGSOFT Softw. Eng. Notes*, 22(6):361–377, Nov. 1997.

[18] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 206–212, New York, NY, USA, 2005. ACM.

[19] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 42, Washington, DC, USA, 2000. IEEE Computer Society.

[20] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *CGO*

'07: Proceedings of the International Symposium on Code Generation and Optimization, pages 185–197, Washington, DC, USA, 2007. IEEE Computer Society.

[21] D. Chen, N. Vachharajani, R. Hundt, S.-w. Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng. Taming hardware event samples for fdo compilation. In *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 42–52, New York, NY, USA, 2010. ACM.

[22] H. Chen, W.-C. Hsu, J. Lu, P.-C. Yew, and D.-Y. Chen. Dynamic trace selection using performance monitoring hardware sampling. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 79–90, Washington, DC, USA, 2003. IEEE Computer Society.

[23] T. Y. Chen and M. F. Lau. Dividing strategies for the optimization of a test suite. *Inf. Process. Lett.*, 60(3):135–141, Nov. 1996.

[24] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo. Testtube: a system for selective regression testing. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 211–220, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[25] S. Cornett. Bullseye testing technology: Minimum acceptable code coverage. http://www.bullseye.com/minimum.html.

[26] T. Dey, W. Wang, J. Davidson, and M. L. Soffa. Characterizing multi-threaded applications based on shared-resource contention. In *IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2011.

[27] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *Proc. of 15th ISSRE*, pages 113–124, 2004.

[28] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 102–112. ACM Press, 2000.

[29] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, 2002.

[30] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proc. of 23rd ICSE*, pages 329–338, 2001.

[31] S. Eranian. Perfmon2. http://perfmon2.sourceforge.net.

[32] ewing. Automated Testing on Mobile Devices for iOS and Android. http://blog.anscamobile.com/2011/08/automated-testing-on-mobile-devices-part1/.

[33] R. F. Fischer, K. and A. Chrusckicki. A methodology for retesting modified software. *In Proceedings of the National Tele. Conference B-6-3*, pages 1–6, Nov. 1981.

[34] S. Forrest. Genetic algorithms. In A. B. Tucker, editor, *The Computer Science Handbook*. CRC Press, Boca Raton, FL, second edition, June 2004.

[35] P. Frankl and E. Weyuker. An applicable family of data flow testing criteria. *Software Engineering, IEEE Transactions on*, 14(10):1483–1498, Oct 1988.

[36] M. Garey and D. Johnson. *Computers and Intractability: A Guide to NP-Completeness*. W.H. Freeman and Company, San Fransisco, 1979.

[37] D. E. Goldberg. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. Addison-Wesley, Reading, MA, 2002.

[38] D. E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3(5):493–530, 1989.

[39] E. Gossett. *Discrete Mathematics with Proof*. Pearson Education, Inc., Upper Saddle River, New Jersey, 2003.

[40] M. Harman. The current state and future of search based software engineering. In *Future of Soft. Eng.*, pages 342–357, 2007.

[41] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, July 1993.

[42] M. J. Harrold and M. L. Soffa. An incremental approach to unit testing during maintenance. *In Proceedings of the Conference on Software Maintenance*, pages 362–367, Oct. 1988.

[43] J. Hartmann and D. Robson. Techniques for selective revalidation. *Software, IEEE*, 7(1):31 –36, jan. 1990.

[44] J. Hollingsworth, O. Niam, B. Miller, Z. Xu, M. Goncalves, and L. Zheng. Mdl: a language and compiler for dynamic program instrumentation. In *Parallel Architectures and Compilation Techniques., 1997. Proceedings., 1997 International Conference on*, pages 201 –212, nov 1997.

[45] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[46] IBM. Rational pure coverage. http://www-01.ibm.com/software/rational/.

[47] Intel Corporation. *Intel 64 and IA-32 Architectures Software and Developer's Manual, Volumes 3A and 3B*. Intel Corporation, Santa Clara, CA, USA, March 2010.

[48] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. In *Proceedings of the IEEE International Conference on*

Software Maintenance (ICSM'01), ICSM '01, pages 92–, Washington, DC, USA, 2001. IEEE Computer Society.

[49] G. M. Kapfhammer. Software testing. In A. B. Tucker, editor, *The Computer Science Handbook*. CRC Press, Boca Raton, FL, second edition, June 2004.

[50] G. M. Kapfhammer, M. L. Soffa, and D. Mosse. Testing in resource constrained execution environments. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 418–422, 2005.

[51] E.-A. Karlsson, L.-G. Andersson, and P. Leion. Daily build and feature development in large distributed projects. *Proc. of 22nd ICSE*, pages 649–658, 2000.

[52] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer-Verlag, Berlin, Germany, 2004.

[53] M. Kessis, Y. Ledru, and G. Vandome. Experiences in coverage testing of a Java middleware. In *Proc. of 5th SEM*, pages 39–45, 2005.

[54] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proc of 24th ICSE*, pages 119–129, 2002.

[55] K. Koster and D. Kao. State coverage: a structural test adequacy criterion for behavior checking. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers*, ESEC-FSE companion '07, pages 541–544, New York, NY, USA, 2007. ACM.

[56] H. Labs. Overview of perfmon kernel interface. http://www.hpl.hp.com/research/linux/perfmon/perfmon.php.

[57] M. Larabel. Linux 3.4 kernel will gain more android patches. http://www.phoronix.com/scan.php?page=news_item&px=MTA2ODA.

[58] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 282 –290, nov 1992.

[59] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[60] A. R. Lebeck and D. A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *Computer*, 27:15–26, 1994.

[61] Y. Lei and J. Andrews. Minimization of randomized unit test cases. In *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, pages 10 pp. –276, nov. 2005.

[62] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 417–420, New York, NY, USA, 2007. ACM.

[63] H. K. N. Leung and L. J. White. A study of integration testing and software regression at the integration level. In *Proceedings of Conference on Software Maintenance*, pages 290–300, Nov. 1990.

[64] J. Levon. OProfile - A System Profiler for Linux. `http://oprofile.sourceforge.net`.

[65] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Trans. on Soft. Eng.*, 33(4):225–237, 2007.

[66] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. *SIGPLAN Not.*, 38:141–154, May 2003.

[67] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. *SIGPLAN Not.*, 40:15–26, June 2005.

[68] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. *SIGARCH Comput. Archit. News*, 34(5):37–48, 2006.

[69] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[70] J. Mars and R. Hundt. Scenario based optimization: A framework for statically enabling online optimizations. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 169–179, Washington, DC, USA, 2009. IEEE Computer Society.

[71] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. Contention aware execution: online contention detection and response. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 257–265, New York, NY, USA, 2010. ACM.

[72] P. McNamee and M. Hall. Developing a tool for memoizing functions in C++. *ACM SIGPLAN Not.*, 33(8):17–22, 1998.

[73] A. Memon, I. Banerjee, N. Hashmi, and A. Nagarajan. DART: A framework for regression testing "nightly/daily builds" of GUI applications. In *Proc. of ICSM*, 2003.

[74] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *Computer*, 28(11):37 –46, nov 1995.

[75] J. Misurda, B. R. Childers, and M. L. Soffa. Jazz2: a flexible and extensible framework for structural testing in a java vm. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 81–90, New York, NY, USA, 2011. ACM.

[76] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa. Demand-driven structural testing with dynamic instrumentation. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 156–165, New York, NY, USA, 2005. ACM.

[77] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, 1998.

[78] I. Moore. Jester- a JUnit test tester. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, pages 84–87, May 2001.

[79] A. J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage-based test sets. In *In Proc. Twelfth Int'l. Conf. Testing Computer Softw*, pages 111–123, 1995.

[80] T. Ostrand and E. Weyuker. Using dataflow analysis for regression testing. In *Proceedings of the Sixth Annual Pacific Northwest Conference on Software Quality*, pages 233–247, sept 1988.

[81] W. Perry. *Effective Methods for Software Testing*. John Wiley & Sons, Inc., New York, New York, 1995.

[82] B. Pettichord. Seven steps to test automation success. In *Proc. of STAR*, 1999.

[83] D. Pisinger. Core problems in knapsack algorithms. *Operations Research*, 47(4):570–575, 1999.

[84] C. Poole and J. W. Huisman. Using extreme programming in a maintenance environment. *IEEE Softw.*, 18(6):42–50, 2001.

[85] V. Ramasamy, R. Hundt, W. Chen, and D. Chen. Feedback-directed optimizations with estimated edge profiles from hardware event sampling. In *Open64 Workshop at CGO 2008*, Boston, MA, USA, 2008. ACM.

[86] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[87] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering Methodology*, 6(2):173–210, 1997.

[88] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *In Proceedings of the International Conference on Software Maintenance*, pages 34–43, 1998.

[89] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929 –948, oct 2001.

[90] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *Proceedings of the International Conference on Software Maintenance*, Oxford, September 1999.

[91] V. Roubtsov.

[92] P. Runeson. A survey of unit testing practices. *IEEE Softw.*, 23(4):22–29, July 2006.

[93] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *Proc. of 20th ASE*, pages 114–123, 2005.

[94] D. Saff and M. D. Ernst. An experimental evaluation of continuous testing during development. In *Proc of ISSTA*, pages 76–85, 2004.

[95] R. Santelices and M. J. Harrold. Efficiently monitoring data-flow test coverage. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 343–352, New York, NY, USA, 2007. ACM.

[96] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W. Chen, and W. Zheng. Racez: a lightweight and non-invasive race detection tool for production applications. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 401 – 410, may 2011.

[97] A. Shye, M. Iyer, V. J. Reddi, and D. A. Connors. Code coverage testing using hardware performance monitoring support. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 159–163, New York, NY, USA, 2005. ACM.

[98] A. M. Smith, J. Geiger, G. M. Kapfhammer, and M. L. Soffa. Test suite reduction and prioritization with call trees. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 539–540, New York, NY, USA, 2007. ACM.

[99] A. M. Smith and G. M. Kapfhammer. An empirical study of incorporating cost into test suite reduction and prioritization. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 461–467, New York, NY, USA, 2009. ACM.

[100] M. L. Soffa, K. Walcott, and J. Mars.

[101] M. Spinelli. Testing memory usage on mobile safari. http://cubiq.org/testing-memory-usage-on-mobile-safari.

[102] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proc. of ISSTA*, pages 97–106, 2002.

[103] M. M. Systems. Java code coverage analyzer - jCover. http://www.mmsindia.com/JCover.html.

[104] A.-B. Taha, S. Thebaut, and S.-S. Liu. An approach to software fault localization and revalidation based on incremental data flow analysis. In *Computer Software and Applications Conference, 1989. COMPSAC 89., Proceedings of the 13th Annual International*, pages 527 –534, sept 1989.

[105] S. Tallam and N. Gupta. A concept analysis inspired greedy algorithm for test suite minimization. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '05, pages 35–42, New York, NY, USA, 2005. ACM.

[106] TestCocoon Software. TestCocoon - Code Coverage Tool for C/C++ and C#. http://doc.froglogic.com/squish-coco/2.0/codecoverage.html.

[107] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. *SIGSOFT Softw. Eng. Notes*, 27(4):86–96, July 2002.

[108] R. Torkar and S. Mankefors. A survey on testing and reuse. In *Proceedings of the IEEE International Conference on Software-Science, Technology & Engineering*, SWSTE '03, pages 164–, Washington, DC, USA, 2003. IEEE Computer Society.

[109] L. Torvalds. Linux 3.4-rc1. https://lkml.org/lkml/2012/3/31/214.

[110] A. Tran, M. Smith, and J. Miller. A hardware-assisted tool for fast, full code coverage analysis. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 321 –322, nov. 2008.

[111] G.-R. Uh, R. Cohn, B. Yadavalli, R. Peri, and R. Ayyagari. Analyzing dynamic binary instrumentation overhead. In *Workshop on Binary Instrumentation and Application*, 2007.

[112] J. M. Voas. PIE: a dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–735, 1992.

[113] F. I. Vokolos and P. G. Frankl. Pythia: a regression test selection tool based on textual differencing. In *IFIP TC5 WG5.4 3rd internatinal conference on on Reliability, quality and safety of software-intensive systems*, ENCRESS '97, pages 3–21, London, UK, UK, 1997. Chapman & Hall, Ltd.

[114] K. R. Walcott. Prioritizing regression test suites for time-constrained execution using a genetic algorithm. Technical Report CS05-11, Department of Computer Science, Allegheny College, Meadville, PA, 2005.

[115] K. R. Walcott, G. M. Kapfhammer, R. S. Roos, and M. L. Soffa. Time-aware test suite prioritization. In *ISSTA06: Proceedings of the International Symposium on Software Testing and Analysis*, Portland, Maine, USA, July 2006.

[116] K. R. Walcott-Justice, G. M. Kapfhammer, and M. L. Soffa. Prioritizing test suites for time constrained execution. In *Journal Paper: To be submitted.*

[117] K. R. Walcott-Justice, J. Mars, and M. L. Soffa. THeME: A System for Testing by Hardware Monitoring Events. In *ISSTA 2012: Proceedings of the International Symposium on Software Testing and Analysis*, July 2012.

[118] E. J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Commun. ACM*, 31(6):668–675, June 1988.

[119] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th international conference on Software engineering*, ICSE '95, pages 41–50, New York, NY, USA, 1995. ACM.

[120] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini. Test set size minimization and fault detection effectiveness: a case study in a space application. *J. Syst. Softw.*, 48(2):79–89, Oct. 1999.

[121] Q. Yang, J. J. Li, and D. Weiss. A survey of coverage based testing tools. In *Proceedings of the 2006 international workshop on Automation of software test*, AST '06, pages 99–103, New York, NY, USA, 2006. ACM.

[122] C. Yilmaz and A. Porter. Combining hardware and software instrumentation to classify program executions. In *Proceedings of the 2010 Foundations of Software Engineering Conference*. ACM, 2010.

[123] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *Proc. of ISSTA*, pages 140–150, 2007.

[124] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.

[125] B. Zorman, G. M. Kapfhammer, and R. S. Roos. Creation and analysis of a Javaspace-based genetic algorithm. In *Proceedings of the 8th International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, June 2002.