EFFICIENT BRANCH AND NODE TESTING

by

Jonathan Misurda

Bachelor of Science, University of Pittsburgh, 2003

Master of Science, University of Pittsburgh, 2005

Submitted to the Graduate Faculty of

the Department of Computer Science in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2011

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Jonathan Misurda

It was defended on

November 29, 2011

and approved by

Bruce R. Childers, Department of Computer Science

Panos Chrysanthis, Department of Computer Science

Mary Lou Soffa, University of Virginia, Department of Computer Science

Youtao Zhang, Department of Computer Science

Dissertation Director: Bruce R. Childers, Department of Computer Science

EFFICIENT BRANCH AND NODE TESTING

Jonathan Misurda, PhD

University of Pittsburgh, 2011

Software testing evaluates the correctness of a program's implementation through a test suite. The quality of a test case or suite is assessed with a coverage metric indicating what percentage of a program's structure was exercised (covered) during execution. Coverage of every execution path is impossible due to infeasible paths and loops that result in an exponential or infinite number of paths. Instead, metrics such as the number of statements (nodes) or control-flow branches covered are used.

Node and branch coverage require instrumentation probes to be present during program runtime. Traditionally, probes were statically inserted during compilation. These static probes remain even after coverage is recorded, incurring unnecessary overhead, reducing the number of tests that can be run, or requiring large amounts of memory.

In this dissertation, I present three novel techniques for improving branch and node coverage performance for the Java runtime. First, Demand-driven Structural Testing (DDST) uses dynamic insertion and removal of probes so they can be removed after recording coverage, avoiding the unnecessary overhead of static instrumentation. DDST is built on a new framework for developing and researching coverage techniques, Jazz. DDST for node coverage averages 19.7% faster than statically-inserted instrumentation on an industry-standard benchmark suite, SPECjvm98.

Due to DDST's higher-cost probes, no single branch coverage technique performs best on all programs or methods. To address this, I developed Hybrid Structural Testing (HST). HST combines different test techniques, including static and DDST, into one run. HST uses a cost model for analysis, reducing the cost of branch coverage testing an average of

48% versus Static and 56% versus DDST on SPECjvm98.

HST never chooses certain techniques due to expensive analysis. I developed a third technique, Test Plan Caching (TPC), that exploits the inherent repetition in testing over a suite. TPC saves analysis results to avoid recomputation. Combined with HST, TPC produces a mix of techniques that record coverage quickly and efficiently.

My three techniques reduce the average cost of branch coverage by 51.6–90.8% over previous approaches on SPECjvm98, allowing twice as many test cases in a given time budget.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

PREFACE

This dissertation is dedicated to Sandra "Sandy" Scharding (July 1, 1981 – May 13, 2011). Without Sandy's love and support, none of this would be possible. The incredible strength she showed living with Cystic Fibrosis inspires me daily.

I also wish to thank my parents for all of their help through these years. Their love and support also made this possible.

Finally, I am thankful for the patience of my committee, the department, and my students.

1.0 INTRODUCTION

Beware of bugs in the above code; I have only proved it correct, not tried it.

—Donald Knuth, March 29, 1977

As programs grow ever larger and more complex, demonstrating them to be bug-free becomes an increasing challenge. In a 2002 study, the National Institute of Standards and Technology estimated that errors in software cost the United States $59.5 billion each year [50]. With so much money at stake, not to mention issues of health and safety, it is clear that reducing the number of bugs found in software is a paramount concern of the software development process.

For example, in October 2011, British car manufacturer Jaguar Cars Ltd. issued a recall of nearly 18,000 cars due to a software bug that prevented the cruise control from being disabled. It could only be disengaged by shutting off the ignition, something not safely done at speed. Jaguar stated that "in some circumstances the cruise control may not respond to the normal inputs" [30], a bug that adequate testing should be able to expose.

However, the size of modern programs can make the task of testing nearly impossible. Larger programs have more source code statements, more branches, and more data values where a mistake may occur. In a 1999 empirical study, Rothermel et al. report that the software product of an industry partner containing about 20,000 lines of code requires seven weeks to run the entire test suite [59].

Traditionally there have been two schools of Software Engineers: those who want to formally verify a program through proofs, and those who want to test the actual behavior

of the code's execution versus its expected behavior. Formal verification is a lofty goal, and while automatic theorem provers have been somewhat successful—usually with specially designed programming languages or annotations—the techniques are far from general. Worse, they are often impractical for the very large programs that need it the most [65, 28]. Instead, through the intelligent use of test techniques, software testing can be made practical and accurate.

1.1 SOFTWARE TESTING

Just as most people will test drive a car before they purchase it, even novice programmers instinctively check program output to determine the behavior of a program. However, this is usually done in an *ad hoc* fashion. The bigger question that may be asked is: "Can this be done in a systematic way?" The answer, of course, is a resounding "Yes!". Software Testing is the discipline of Software Engineering devoted to empirically testing a program's execution to determine software correctness [38].

Software testing begins with the realization that to effectively check a program's behavior, one needs to test as much of the code as possible. Too many programs have been written where "rare" code, such as error handlers, have been left untested. The first step then is to construct a set of *test cases* that are designed to exercise the program via carefully constructed inputs. The second step is to run the *suite* of test cases and to check the output. Any discrepancy between the actual and expected results constitutes a bug.

This strategy can be extremely effective, but it hinges on the tests being created in a deliberate fashion rather than just *ad hoc* collections of input. There needs to be a way to assess the quality of a test suite—i.e., how well it tests the program. One possible way of evaluating the test suite is to look at the structure of a program—its control and data flow paths—and to count how many of those paths are taken versus how many exist.

1.2 STRUCTURAL TESTING

While the ideal in *Structural Testing* is to completely test every path a program can take, this is an untenable goal in practice. Some portions of program code may be infeasible, and no test case will ever execute the code. Even if the tests are restricted to feasible paths, the number of possible paths in a program that has loops may be vast or even infinite [37].

If path-based structural testing could be done simply, the evaluation of a test suite would be the ratio of executed paths to the total number of possible paths. The lower the percentage of covered paths, the less thorough the test suite. This ratio is a *coverage metric*, where a test case is said to *cover* some path or paths.

Since path-based structural testing is hard or impossible in many cases—specifically those large programs that need thorough testing—some other, more practical manner by which to determine the quality of a test suite is needed. One possible approach is to relax the granularity of the structure from a path to something simpler to test such as individual code statements or branches.

1.2.1 The Coverage Metrics

There are two classes of coverage metrics: the control flow metrics and the data flow metrics. Zhu et al. describe these tests formally [69] and I will describe them briefly here for convenience. The most commonly-used control flow metrics are:

Path Coverage Path coverage is the ideal coverage metric. If a test suite covers 100% of the paths of a program, including implied paths such as integer overflow or other exceptions, the program should be bug-free. There are many implicit assumptions, however, including the questions of what to do with infeasible code, what to do when inputs cause a program to never halt, or how possible it is to actually generate all possible inputs to get all paths.

Branch Coverage While there may be an infinite number of paths due to back-edges, there are a finite number of branch instructions. Branch coverage seeks to evaluate

a test suite for how many branches it covers out of the total number of branches in the code. This coverage metric is, however, weaker than path coverage, since it is possible to construct a path that is comprised of covered branches but has other paths uncovered.

Node Coverage Node coverage (also called statement coverage), he weakest of all the control flow coverage metrics, simply seeks to discover if every statement in the program has been executed by a test suite. Since the path to the node is not even recorded at the branch level, statement coverage can miss bugs that result from two nodes being executed in some order, even though both nodes could have been covered.

The data flow coverage metrics include:

Def-Use Coverage A data flow metric that seeks to determine how many Definition-Use (DU) pairs of a variable have been covered out of all possible ones. A DU-pair is an edge between where a variable is defined and where that assigned value is used.

All Uses Coverage The all-uses coverage metric records if for every variable definition, a path has been covered that reaches every use. This is weaker than the All DU-pairs criterion since only a single path must be exercised instead of every path.

All Defs Coverage This metric records coverage of every definition of a variable that reaches a use. It is the weakest of the data-flow coverage metrics.

The subsumption hierarchy for the control flow metrics is that path coverage subsumes branch coverage which subsumes node coverage. For data flow metrics, Def-Use coverage subsumes All Uses which subsumes All Defs. Many other works focus on the creation of test cases and specific coverage metrics [43, 26, 38, 29].

1.3 THE CHALLENGES OF EFFICIENT STRUCTURAL TESTING

With such a well-defined testing strategy, why has coverage testing not become standard practice among all large software projects? For some reason, the discipline that software testing requires often falls to the wayside during software projects. Often programmers

are unmotivated to write test suites, thinking that the extra work it involves is wasted rather than saving debugging time down the road. Also, testing is just one more obstacle in the increasingly tight time and monetary budgets of many companies. But the simplest of the many reasons is that currently it is not feasible to do quick structural testing on large programs. This is due to the deficiencies in the current tools and techniques.

Coverage testing needs to be emphasized as a major part of the development process. The easiest way to do this is to integrate testing into the development environment. However, existing test techniques are not automated, and limited to what a tool provides by default, which is often only low-content coverages like statement coverage. Even worse, existing instrumentation techniques used in commercial tools are slow and often require access to the source code. Java testing tools that use the Java Debugging Interface disable Just-In-Time compilation and default back to painfully slow interpretation [7, 44].

As if the slow methods of data gathering were not enough, existing coverage algorithms place instrumentation probes without regard to the actual execution of the program. This means instrumentation is unnecessarily placed in regions of code that are infeasible, wasting time. Frequently these probes are inserted at compile-time, possibly masking bugs such as buffer overruns with the storage for counter variables.

Take for example, the control flow graph (CFG) in Figure 1.1. The program in this figure is a simple one that adds the even and odd integers from one to one million. With traditional tools, static instrumentation would be placed into each basic block (indicated by the colored boxes) and even though every node or branch but the return would be covered on the second iteration, the program would still be incurring high overheads. In the example program of Figure 1.1, there will be 4,000,003 total probe executions when node coverage only needs one execution of the instrumentation probe in each of the seven basic blocks.

Figure 1.2 shows an approach to improving the cost of structural testing that attempts to place less probes and infer coverage. The algorithm was developed by H. Agrawal and uses dominator information to allow one probe to record the coverage of multiple structures [12]. The resulting probe locations shown in the figure yield over a 50% reduction in the static number of probes. However, when the code is executed, there still

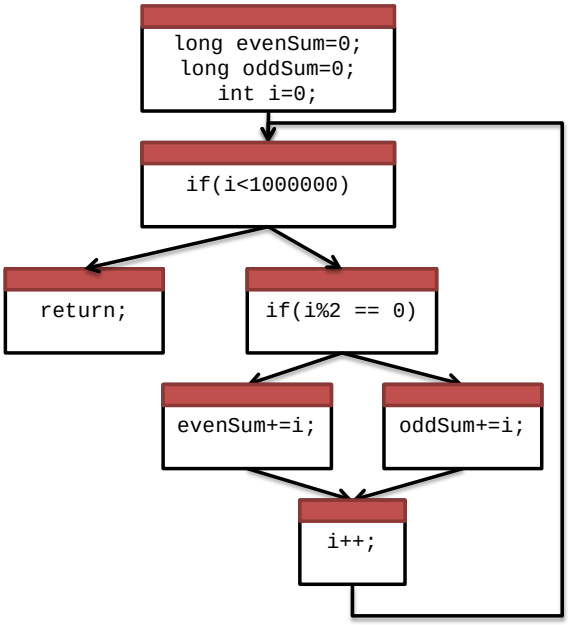

Figure 1.1: Static instrumentation is not removed after recording coverage.

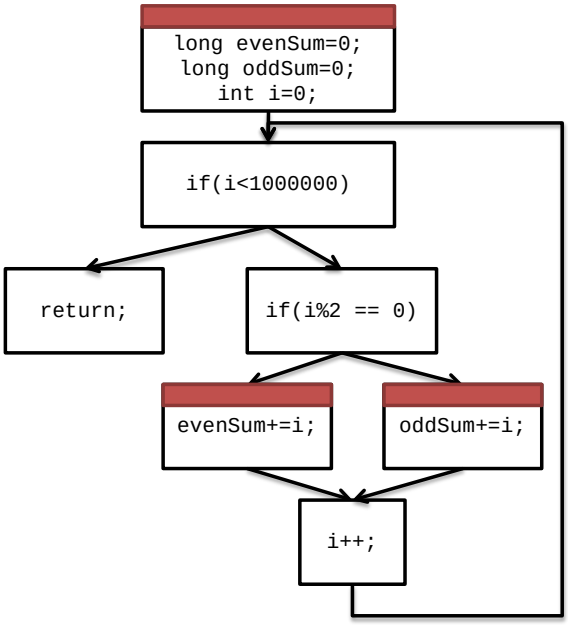

Figure 1.2: Agrawal's algorithm statically places fewer instrumentation probes.

remain 1,000,001 dynamic probe executions when only three were necessary to record full coverage. Additionally, the cost of computing the probe locations is high, reducing the advantage of the fewer probe executions.

1.4 RESEARCH OVERVIEW

With these shortcomings, it is clearly time for a better system for doing structural testing. This dissertation presents an end-to-end solution efficiently collecting coverage information for node (statement) and branch coverage in a Java JVM that uses just-in-time (JIT) compilation.

First, I present a new framework, called Jazz, which allows for the easy implementation and research of different structural test techniques. Jazz supports inserting multiple types of instrumentation at different phases in the JIT compilation process. It does the insertion by interfacing into the Java JIT compilation process and exposing an interface for building *test planners* that drive the placement and operation of instrumentation probes. Additionally, Jazz provides facilities to handle memory management and control-flow analysis. It also provides a test specification language that allows for convenient specification of which test techniques to apply to which packages, classes, and methods in a particular testing run. On top of these services, I developed an extensible test library that formed the foundations for the novel test techniques this dissertation will present.

One member of the test library is a novel technique for structural testing called *demand-driven structural testing* (DDST). DDST uses instrumentation probes constructed out of *fast breakpoints* [40] that can be easily inserted and removed during the execution of the tested program. Dynamic insertion and removal of probes allows instrumentation to have as short of a lifetime as possible, taking advantage of the fact that coverage is a boolean property that, in practice, converges quickly. By removing instrumentation as soon as it is no longer necessary, the majority of the tested program's execution will be instrumentation-free. Experiments using DDST show it to be the best choice for node coverage and is a good choice for loop-intensive code when testing with branch coverage.

However, uniform application of DDST-based branch coverage testing to all programs or even all methods in a program is not always the best choice when compared to mixing in more traditional static, always-present instrumentation. DDST requires a more expensive instrumentation probe and some methods have a control-flow structure called a *stranded block*, which prevents instrumentation from being removed when coverage has not reached 100%. Instead of using just one test technique, this dissertation presents a hybrid approach called Profile-driven Hybrid Structural Testing (HST) that combines three different branch coverage test techniques: DDST, Static, and Static with Agrawal's optimization technique. Information from a profiling run on a small input is combined with a model of instrumentation probe behavior to provide a customized plan of testing techniques to be applied to the methods of a program. Results show that the hybrid approach improves upon the naïve application of a single test technique by an average of 48% versus Static and 56% versus DDST.

Structural testing can be further improved by examining the software testing process as a whole. In general, there is a suite of more than one test case over which the program is run. Since there is repeated execution of methods that have not changed between runs, some of the cost of planning where and how to instrument the program can be cached and reloaded on subsequent test cases from the suite. This dissertation introduces Test Plan Caching (TPC) that uses saved test plans. By eliminating the unnecessary recomputation of test plans, more test cases out of the suite can be executed in a limited time budget.

1.5 CONTRIBUTIONS

This dissertation makes a number of contributions to the challenges of structural testing. They are:

1. A formalized notation and model of structural testing that captures the properties inherent in the various test techniques, allowing for analysis and development of new techniques,

2. A framework, Jazz, for the convenient implementation and evaluation of structural testing algorithms,
3. Testspec, a language for the specification of regions on which to apply the structural tests developed in Jazz,
4. A novel approach for structural testing using demand-driven instrumentation (DDST),
5. Hybrid Structural Testing (HST), a profile-based approach for selecting the appropriate branch test technique for use on a given method, and
6. Test Plan Caching (TPC), a method for using repeated test runs over the test cases of a suite to further improve testing performance by caching over the entire suite.

1.6 ASSUMPTIONS & SCOPE

This work addresses the problem of efficient structural testing under a set of assumptions. First, the Jazz framework targets a Java Virtual Machine (JVM) that uses just-in-time compilation. The JVM must be freely modifiable to allow for the addition of the Jazz framework components. Secondly, I focus on baseline compilation where aggressive optimizations that would result in code movement or elimination are not applied. The techniques of this dissertation apply to arbitrary sequences of binary code, but with optimization enabled, mapping the coverage information back to the source code becomes more difficult. The same problem has been solved with debuggers in the work of Jaramillo [35] and those techniques could be applied to DDST.

The Java programming language also makes it possible to compute a control flow graph before the code is run, as there are no computed branches with targets unknown at JIT compilation time.

The instrumentation is written as self-modifying x86 machine code. It is assumed that self-modifying code is supported by the hardware and the operating system.

This work targets single-threaded applications written solely in Java with no native code. The modifications to the techniques to support multithreading are addressed in this dissertation, but implementation is future work.

1.7 ORGANIZATION

The rest of this thesis is organized as follows. Chapter 2 examines the background and related work. Chapter 3 presents a new framework, Jazz, designed to facilitate the research and development of structural tests in a JVM. Chapter 4 describes a novel technique for doing structural testing using instrumentation that is dynamically inserted and removed during program execution. Experimental results from testing DDST techniques show that none of the evaluated techniques for branch coverage is consistently the best choice across all programs, nor even across all methods in a single program. Chapter 5 thus presents a way of using a small profiling input to determine which technique to apply to a method on subsequent full testing runs. In Chapter 6, I seek to exploit the testing process to identify ways where a test suite can be optimized beyond a single test case run by using caching and information gathered in prior runs of the test suite. Finally, Chapter 7 concludes and discusses future work.

2.0 BACKGROUND AND RELATED WORK

In this chapter, I present the background of software testing and discuss previous work to frame the new solutions presented in the remainder of this dissertation. Related work is presented in four parts. First, existing coverage tools are discussed. Next, previously suggested techniques for improving the cost of coverage collection are given. Third are several instrumentation techniques useful for software testing. Finally, languages related to specifying how to test or where to instrument are explored.

2.1 SOFTWARE TESTING

Software testing is the process of assessing the functionality and correctness of a program through execution or analysis [38]. Its purpose is to reveal software *faults* before they manifest themselves as *failures* during regular program usage. A failure is defined to be "the inability of a system or component to perform its required functions within specified performance requirements" [33]. A fault is a portion of a program's source code that is incorrect [33].

The discovery of faults through testing is, at its heart, an intractable problem. For instance, the number of inputs to a program may be exponential in size, an input may cause a program to never terminate, or the fault may lie in the specification of the problem rather than the implementation of the program [18, 38]. However, this does not mean that testing should be ignored.

Testing can be done by inspection or by execution [38]. This dissertation focuses on execution-based testing approaches. This means that the program to be tested is run with

an input designed to exercise the program in a certain way, known as a *test case*. Since a single run may not exercise the program adequately, there may be multiple test cases assembled into a *suite* [37, 38, 33, 18, 69].

However, feedback is necessary to know how well a test suite has exercised the program, and thus exposed faults. For this, an *adequacy criterion* is used [69]. A question then is: "What constitutes a reasonable measure of adequacy?" Two possible categories are criteria that are *specification-based*, where testing determines if the software meets a set of predefined requirements, and *program-based* where the adequacy of testing is determined in relation to the properties of the program under test [69].

2.1.1 Structural Testing

This dissertation focuses on a program-based adequacy criterion, specifically *structural testing*, where the requirements for a test suite are based upon a particular aspect of the structure of the program. Structures that can be explored include *control-flow* ones, such as the nodes, edges, or paths of a program's control flow graph, or *data-flow* ones, which track variable definitions and uses [69].

Not all adequacy criteria are equal. For instance, if a test case exercises a control-flow structure such as a path through a program, that path is said to be *covered*. Coverage of a path implies that each branch taken as part of that path is also covered. In this way, a criterion such as *path coverage* is said to *subsume* the *branch coverage* criterion [69, 38].

Clarke et al. created and proved a subsumption hierarchy that relates data- and control-flow adequacy criteria to each other. The relevant part of this hierarchy for this dissertation is shown in Figure 2.1 [23]. Path coverage is the most complete of the criteria as covering all paths would be to test all possible executions of the program. However, the number of paths can be infinite or exponential in programs with loops [69, 38]. Additionally, some paths are infeasible, where no input will ever exercise it [49].

For the data-flow criteria, there are both definitions of variables and uses of the variables to cover (*du coverage*). A use of a variable in a computation is called a *c-use* and a use of a variable as a predicate in a conditional statement is known as a *p-use* [69, 38].

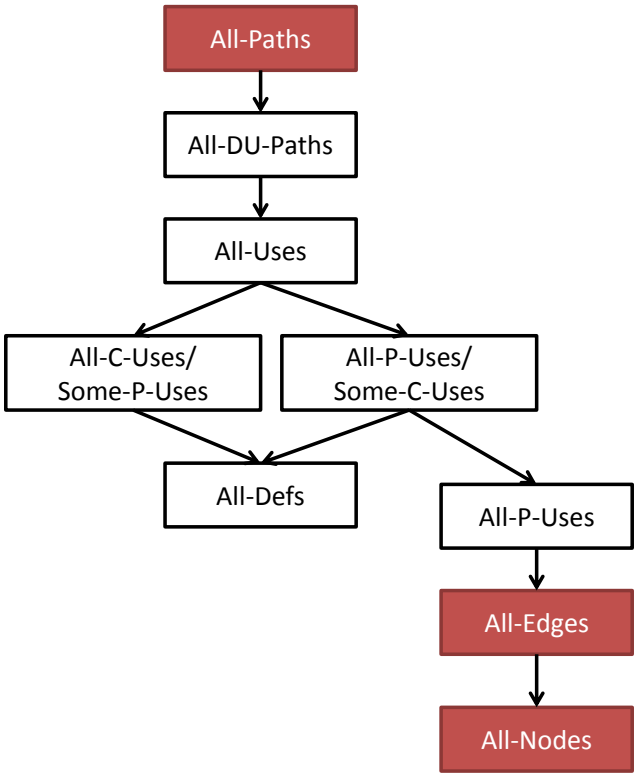

Figure 2.1: Partial subsumption hierarchy for adequacy criteria.

Control-flow criteria are based on paths, branches, and *nodes*—the basic blocks of the control flow graph, which correspond to the statements in the program's source code. The control-flow criteria are colored in Figure 2.1. This dissertation focuses on branch and node coverage.

2.1.2 Uses of Structural Coverage Criteria

The original motivation for collecting coverage with structural testing was to determine test suite adequacy. However, coverage results can also be used to drive the composition and application of the test suite to a program under test. Additionally, it can be used as part of *regression testing* to select the appropriate test cases to target where new modifications of the source code might have introduced a new fault into the program [38].

Test selection uses the results of coverage information to run a subset of test cases out from a suite that target the locations where code has been modified. This is done to reduce the cost of regression testing [16, 56, 38].

Some test cases in the test suite are more useful than others. These cases can be chosen to run first to make sure the maximal amount of coverage can be achieved in a given time budget. *Test suite prioritization* also allows faults to be exposed sooner in the testing process. Structural coverage can be used to assess the priority of a test case [57, 59, 29, 38].

In addition to prioritization, *test suite reduction* uses coverage results to prune test cases that do not supply additional information from the other cases in the suite [36, 58].

2.2 RELATED WORK

Due to the well-established utility of coverage testing, there is a body of traditional tools and techniques to collect coverage. This section describes those tools and techniques developed to more quickly collect coverage and other similar runtime properties. The techniques below and in this thesis also depend upon program instrumentation. Finally, I examine previous work regarding the problem of specifying where in a program to test.

2.2.1 Coverage Tools

Commercial tools such as JCover [7], Clover [3], and IBM's Rational Suite [32] provide various methods to do node and occasionally branch coverage, as part of an integrated tool. When applied to Java programs they avoid instrumenting the code by using the Java Virtual Machine (JVM) interface meant for external debuggers. When this interface is used, Just-In-Time compilation is disabled to use the much slower interpretation mode.

Open source tools such as Emma [6] and Cobertura [4] take a different approach. They often offer two separate ways to collect coverage information. The first approach is to use a tool such as the Bytecode Engineering Library (BCEL) [2] to rewrite a Java class file to insert instrumentation. A second approach involves using a custom classloader that inserts the instrumentation when a JVM requests a class the first time.

2.2.2 Improving Coverage Testing Performance

Ball and Laurus seek to improve the performance of path profiling, which counts the number of times a particular path has been executed [17]. From this information, obviously coverage can be inferred. Path profiling is useful to determine the "hot" paths of a program in an effort to focus attention from an optimizing compiler or runtime system. Their technique generates for each path a unique sum that is computed by additions that occur along each branch. This reduces the instrumentation to little more than an `add` instruction, which is likely as lightweight instrumentation as is possible for this process.

Arnold and Ryder improve upon the performance of path profiling where precise execution profiles are not required [14]. Their approach uses sampling, where instrumentation only records execution with some predetermined frequency. Their technique duplicates the region to be instrumented, leaving a clean copy and an instrumented copy. At the start of each region a check of the sampling condition is inserted, and the instrumented or non-instrumented portion is chosen at runtime.

The use of path profiling for recording coverage information does not make sense. In the Ball and Laurus work, the instrumentation, though individual points are efficient, remains much longer than a coverage tool needs to collect its information. Arnold-Ryder

instrumentation acknowledges the idea that most of the time, users want to be running uninstrumented code. However, the very notion of sampling makes it inappropriate for coverage.

Agrawal seeks to improve the performance of node and edge coverage by reducing the total number of locations that need to be instrumented in a program [12]. By computing the pre- and post-dominators in a control flow graph, coverage of certain nodes in the graph imply the coverage of the "dominated" blocks [49]. Experimental results show a significant reduction in the number of locations instrumentation needs to be placed, with only 29% of blocks and 32% of edges being necessary to infer complete coverage in the test programs.

Pavlopoulou and Young attempt to solve the problem of reaching full coverage [53]. They acknowledge that test cases may not expose all feasible paths in a program and wish to use actual deployments of the program in the field to do further testing. However, there is no need to collect coverage information that has already been gathered in the testing phase, so before deployment, the application is statically reinstrumented only in the locations that were not covered during the testing phase of development. The motivation for reducing instrumentation is not to burden the users of production software with the overhead of instrumentation, except where necessary to provide further feedback on software quality.

Tikir and Hollingsworth use a dynamic technique to insert instrumentation probes for node coverage [67]. These probes are placed at method invocation and remain until a time-periodic "garbage collector" removes them. Since the collector does not immediately remove the unneeded instrumentation, they remain in the program, incurring overhead. To reduce the total number of probes, they use an algorithm also based upon the idea of dominators, inferring coverage where possible. They report overhead of .001% to 237%, with an average slowdown of 136% for C programs. Tikir and Hollingsworth do not compare their approach to the immediate removal of an instrumentation probe when it is no longer necessary.

Chilakamarri and Elbaum seek to improve the performance of coverage testing for Java programs by "disposing" of instrumentation when it is no longer needed [20, 21].

Their technique involves replacing a function call to their collection method with `nop` instructions when the coverage information is recorded. They do this by modifying the Kaffe JVM. Compared to static instrumentation with dominator information, they report a 57% improvement on average. However, they only attempt method and node coverage testing. Additionally, their benchmarks exclude the two SPECjvm98 benchmarks that are most loop intensive.

After my preliminary work [48], Santelices and Harrold sought to improve upon def-use testing by attempting to infer coverage via data flow analysis [60]. While many du-chains in straight-line code can be inferred in their approach, any reasonably sized program yields a high level of uncertainty in their coverage results. It seems counter to the testing philosophy to sacrifice such accuracy for speed, when other techniques can likely make no sacrifices at all.

Another avenue for improving coverage performance is using hardware performance monitoring counters to collect coverage. Shye et al. use the Branch Trace Buffer of the Intel Itanium2 processor to gain information about the last four branches taken in the program. Using dominator information and frequent sampling of the performance counters yields 50–80% of the true node coverage value [61]. However, sampling-based approaches with less than 100% accuracy may miss rare code that a good test suite will aim to cover.

For certain processors, increased branch history sizes and the ability to raise an exception when the buffer is full allow for 100% accuracy. Tran, Smith, and Miller use an embedded Blacking DSP with a 16-entry buffer to record node coverage with 100% accuracy by trapping when the buffer is full. They report an overhead of 8–12% but are unable to run a full desktop benchmark suite for comparison due to the nature of their embedded system [68].

Soffa, Walcott, and Mars examine the current state of software testing via hardware performance counters and find that the Intel Core i7 has a 16-entry branch trace store, however access to it enables a special debug mode on the CPU, resulting in a 25–30$\times$ slowdown. Otherwise, a sample-based approach with the aforementioned drawbacks can be used [62].

2.2.3 Instrumentation Techniques

Instrumentation packages like Dyninst [31] and Pin [10] provide generic frameworks for building probes, but have no specific support for software testing. Much of the focus on instrumentation has been motivated by debugging. Kessler [40] proposed a way to create software breakpoints that avoid trapping into the operating system. The code location to break at is overwritten with a jump to a different portion of the address space where the appropriate payload code is executed. However, these breakpoints were proposed for traditional debugging rather than as dynamic instrumentation for structural testing.

In Kapfhammer's dissertation on testing database-centric applications, the tool he developed may use either a static or dynamic approach to instrumentation [39]. An Aspect-Oriented Programming tool called an *aspect weaver* takes a set of conditions to insert instrumentation creating statically instrumented Java classfiles. He also has a modified classloader that can instrument when Java's dynamic loading mechanism encounters a reference to a class for the first time. In both of these cases, the instrumentation remains throughout the execution of the code.

2.2.4 Test Specification

The challenges of generating test cases, running programs multiple times, and collecting and aggregating the results is tedious enough that work has been done to automate it via a Domain-Specific Language (DSL). Balcer, Hasling, and Ostrand propose a domain specific language to ease the overall testing process [15]. Their language does not solve the problem of specifying where or how to do coverage testing.

Bytecode-level instrumentation is an example of a cross-cutting concern that Aspect-Oriented Programming (AOP) [24] seeks to easily support. In particular, the Join Point Model of AspectJ [1] is a rich language that could be used to specify where instrumentation (aspects) should be woven into existing class files. Rajan and Sullivan extend the Join Point Model to support the targeted specification of Java code regions for testing [54]. Kapfhammer's work also suggests an alternative model for a coverage testing DSL based on AOP.

3.0 A FRAMEWORK FOR NODE AND BRANCH COVERAGE

THERE EXIST MANY TOOLS for collecting coverage metrics on programs but none of them are designed to support the research and development of new structural test techniques [7, 3, 32, 6, 4]. To address this shortcoming, I have created a framework for branch and node coverage testing.

The framework consists of two parts. The first is a formalization of the common elements of structural tests such as branch and node coverage. This formalization describes the location of instrumentation probes, the actions those probes must take, the conditions under which those actions should occur, and the memory (state) necessary for collecting coverage.

In addition to the formalization, I have created an extensible and flexible framework called Jazz that serves as a platform on which to research, develop, evaluate, and use techniques for structural testing. Jazz provides the components and support to realize the structural tests described in my formal notation.

Jazz targets Java programs that are compiled with a just-in-time (JIT) compiler. Java presents a rich environment for efficient structural testing as it has increasingly become an important language for many applications where software correctness is vital for daily operation. As applications have grown in complexity, the time spent in testing has increased dramatically and become more costly. As a language and runtime environment, JIT compilation offers a unique facility for testing to achieve good runtime performance by carefully and flexibly choosing how and where to instrument a program to gather coverage information.

Jazz's extensibility comes from the capability to quickly and simply add new test techniques either by modifying existing ones or building new ones on top of the support

services Jazz provides. The framework is flexible and supports a simple test specification language that allows a user to specify when and where to test, including the possibility of applying multiple tests simultaneously in a single program test run. The framework is also flexible because it is can be easily updated to accommodate implementation changes to the underlying Java JIT and VM.

3.1 FORMAL NOTATION FOR BRANCH AND NODE TESTING

Structural tests such as branch and node coverage share several common features. To build a practical framework that will support various techniques for collecting coverage, I developed a formal notation framework that distills the common aspects test planning and collecting coverage information. This framework assumes that the instrumentation necessary for collecting coverage will be placed in basic blocks rather than an alternative approach such as rewriting branch instructions.

For the region R of a program to test, a control flow graph, $(N, E) = CFG(R)$, that captures the structure of R is necessary. The set $I \subseteq N$ represents the nodes in R that will contain instrumentation probes. The data and actions necessary to record the coverage information of R constitute a *test plan*. The test plan for a region contains:

- A set $S \subseteq I$ *seed locations* where instrumentation probes are placed prior to the execution of region R.
- A set of *probe plans* P, one for each element of I, where the *probe plan* for instrumentation probe P_i contains:
  - A set A of actions to perform on probe execution.
  - A set C of conditions used to determine if an action should be performed.
- A state $M \supseteq \bigcup_{1 \leq j \leq |I|} P_j.C$ to update upon P_i being executed. This state is shared among all probes in a region but each region has a separate state. It will also contain the record of coverage, `Cov`, that is collected as the probes execute.

There are two versions of the framework. In this section, I use the notation to express the data and actions necessary for Static Node and Branch Coverage. In Chapter 4, a second version for testing with dynamic instrumentation is used.

3.1.1 Static Node Coverage

Using this formalized notation, each of the traditional techniques for collecting coverage can be expressed. Static Node Coverage places inline instrumentation at compile time that records the coverage of a node when executed. These instrumentation probes record the coverage of the node they are placed in and remain throughout the lifetime of the program.

For a region R on which to perform Static Node Coverage, the set of nodes N are the locations to instrument ($I = N$). The test plan for R contains:

- The set of seed locations $S = I$, where every node in R will get an instrumentation probe before the region is executed.
- The set of probe plans, P, where each probe plan P_i contains:
  - A set of actions A that contains only the action to record coverage of node i in `Cov[i]` = **true**.
  - A set of conditions, C, on which to perform the actions, where C_i is **true** for all actions.
- The state M contains the $|N|$ boolean values that comprise the `Cov` array.

3.1.2 Static Branch Coverage

Similarly to Static Node Coverage, Static Branch Coverage inserts instrumentation probes at compile-time in the source and sink basic blocks of a control flow edge to collect coverage. These probes also remain throughout the entire lifetime of the program.

To record the edge between the source and the sink probes, a thread-local, region-specific variable is introduced called the *previously hit block*. Thus, when the sink instrumentation probe is executed, it can record the edge from the recorded source to itself.

For a region R, the set of nodes N are again the locations to instrument ($I = N$). The test plan for R contains:

- The set of seed locations $S = I$, where every node in R will get an instrumentation probe before the region is executed.
- The set of probe plans, P, where each probe plan P_i contains:
  - A set of actions A that contains:
    1. An action to identify the previously hit block, b.
    2. An action to record coverage of the edge $b \rightarrow i$ in `Cov[b→i]` = **true**.
    3. An action to set the previously hit block indicator to the current one, $b \leftarrow i$.
  - A set of conditions, C, on which to perform the actions, where C_i is **true** for all actions.
- The state M contains the $|E|$ boolean values that comprise the `Cov` array as well as the space necessary for recording the previously hit block.

3.1.3 Agrawal

Agrawal's techniques for reducing the number of static probe locations by using the dominance relationship to infer nodes or edges can be applied to either of the Static techniques. In each case, applying Agrawal's algorithm only affects the set I of basic blocks which have instrumentation probes. This in turn affects the set of seed locations, S, since it is equal to I.

For Static Agrawal Node, Agrawal's algorithm yields the set of nodes that must be instrumented, and thus seeded, $S = I = Agrawal(CFG(R))$. Static Agrawal Branch, however, returns a set of edges for which coverage must be recorded. Each edge in the set is split it into its (source, sink) pair, and I is the union of all sources and sinks.

The probe plans remain the same for each technique. The state M remains the same but the size of `Cov` is reduced accordingly.

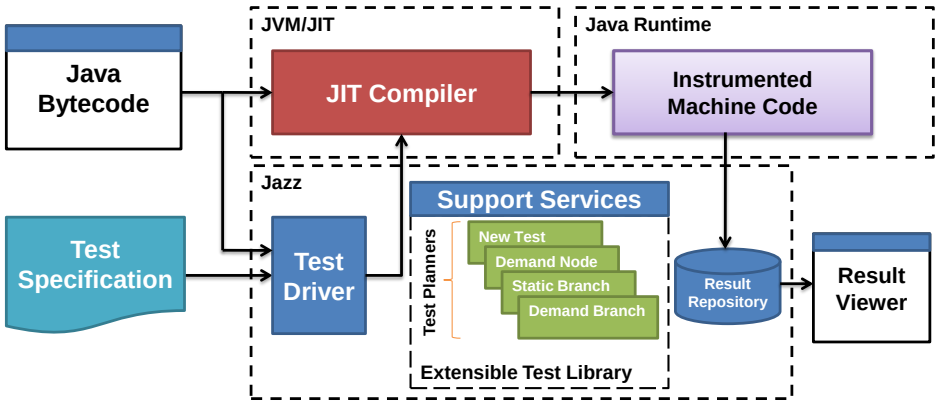

Figure 3.1: The general framework of Jazz.

3.2 JAZZ OVERVIEW

With the formalism of the previous chapter, a software framework to realize various structural test techniques can be built. To this end, I developed Jazz, a framework for creating structural tests for Java programs run under a Java Virtual Machine (JVM) that uses just-in-time (JIT) compilation to convert Java bytecode into native machine code.

An overview of Jazz is shown in Figure 3.1. Jazz integrates with a JVM to provide support for structural testing and is designed to let testing engineers realize a specific structural test for Java through its extensible test library. The implementation of a test technique is called a *test planner* and the data structure that drives runtime probes and records the coverage is a materialized *test plan* from the formalism of the previous section.

Jazz takes as input Java bytecode and a specification of where and how to test Java methods. The *test driver* processes the specification and the bytecode and directs how the instrumented machine code should be produced. The instrumentation monitors program execution to gather coverage results, which are stored for later reporting. When the program terminates, Jazz issues a "coverage report" that is presented to the user.

The core of the Jazz framework is its support services and user-extensible test library of structural test techniques. The support services provide the basic functionality that the

tests need to do instrumentation, control flow analysis, memory allocation, and result collection. Jazz comes with a library of tests that were built using the framework. It also supports additional user-defined tests that extend existing tests or are completely new.

3.2.1 Support Services

Jazz provides support services to simplify building structural tests in the framework. These services are used by the test techniques that are pre-defined in the framework. The services can also be used to construct new tests. The services include:

Instrumentation Jazz provides support for various types of instrumentation including permanent inline instrumentation and transient instrumentation.

Callbacks A specific structural test needs to be informed of events from the JVM, such as when a method is about to be compiled, when a particular bytecode is translated to machine code, when a method has finished compilation, and when the JVM is about to exit. These callbacks serve as the interception points between the JVM and the structural test implementation.

Memory Management Each instantiation of a structural test (i.e., when a method is compiled) needs to allocate memory to do its work and to record information. Jazz supports memory allocation in both the JVM's managed heap and operating system memory buffers (i.e., heap and/or `mmap` space). These memory regions can be used for data or code (marked as executable to support code generation).

Control Flow Analysis Structural tests often need to analyze control flow properties of a method to determine how best to instrument it. Jazz provides analysis support, including basic Control Flow Graph (CFG) generation and more advanced CFG operations such as determining pre- and post-dominator trees, finding strongly connected components, and graph traversal.

3.2.2 Extensible Test Library

The power of Jazz is the ability to add new test techniques with minimal effort. It is designed to be highly extensible with simple facilities. There are two ways to add new

tests: extend an existing test from the test library, or develop a test from scratch (using the support services and a similar technique from the library as a guide or template) and add it to the library.

The initial structural tests that I implemented and added to Jazz's test library allows testing at node (statement) and branch granularity. The test library contains implementations of Static Node and Static Branch coverage testing. It also includes a variation of each that uses a static minimization algorithm developed by H. Agrawal to reduce the amount of static instrumentation needed for coverage testing [12]. These tests are Static Node Agrawal and Static Branch Agrawal. Their implementation is discussed in Section 3.4.

Creating a new test is straightforward as well. If it is a static test, an appropriate existing test in the library can be extended. Alternatively, a new test can be added by extending the superclass for structural testing. The implementation of a new test is simplified by using the support services provided by Jazz.

3.2.3 Test Specification

Jazz is flexible because it offers the capability to specify where and how to test at the method level. It is built with the goal of permitting the combination of different test techniques in a single run of a program, such as choosing to test some methods with branch coverage and others with node coverage, or to instrument methods that are not frequently executed with static instrumentation and the hot path of code with static Agrawal instrumentation. This approach permits fine-grain testing with a single test run, which avoids the expense of multiple independent runs.

To achieve this flexibility, Jazz provides a test specification language, called *testspec*. This language lets a user of the framework specify *rules* about how testing should be performed. Testspec allows for specifying a class and method name to test, as well as the test types to apply to that method. However, it is too tedious to specify every method and its tests. Instead, testspec provides syntax to avoid this burden. Three wildcard operators are included in the specification language.

A special `=all` flag is used to specify all methods in a class. The asterisk globs class

```
spec.benchmarks._209_db.Database:printRec:AGRAWAL_NODE
spec.benchmarks._209_db.Entry:equals:AGRAWAL_NODE
spec.benchmarks._209_db.Database:set_index:AGRAWAL_NODE
spec.benchmarks._209_db.Database:shell_sort:AGRAWAL_NODE
spec.benchmarks._209_db.Database:remove:AGRAWAL_NODE
spec.benchmarks._209_db.Database:getEntry:AGRAWAL_NODE
spec.benchmarks.**:=all:STATIC_BRANCH
```

Figure 3.2: Example test specification for the SPECjvm98 *db* benchmark.

names. The `*` matches any class in a specific package. For instance, `java.util.*` will match `Vector` and `ArrayList` just as it would in a Java import statement. However, when one wishes to test large projects or components, there may be a hierarchy of subpackages to test. Since the asterisk in Java does not import subpackages, it is kept similarly limited in testspec. Instead, the pattern `**` matches any class in the specified package or any subpackage. Thus, the expression `spec.benchmarks.**` is sufficient to select all of the classes in any program in SPECjvm98.

An example of the testspec is shown in Figure 3.2. In the example, the first six lines explicitly state a (class name, method name) pair to be instrumented with the Static Agrawal Node test from the Jazz's test library. The remaining methods that execute at runtime will match the wildcard rule on the last line and be instrumented with the Static Branch test. Any rule that explicitly states a class and method name takes precedence over a wildcard rule. Other rule conflicts are resolved by using the earliest rule specified.

3.3 JAZZ IMPLEMENTATION

Jazz is integrated into the Jikes RVM [13], a just-in-time compiler (JIT) for Java that is itself written in Java. One of the major challenges of having a framework that is coupled to a separate code base, such as the RVM, is developing in an environment that is constantly

changing. The obvious temptation is to take the most recent release version and ignore any updates while developing one's own component. This is not without significant disadvantages, however, as one misses out on any improvements or bug fixes made to the code base since the version you decided to target was released. However, the effort of porting code to new versions may not justify supporting every point release. Nevertheless, new major versions are often worth the effort. In this case, to make the effort as minimal as possible, Jazz was developed to tie into the RVM in as few places as possible.

Jazz's test driver (see Figure 3.1) is implemented by the class `FrameworkHarness`. This class facilitates interaction with the RVM and isolates Jazz from it. I identified four minimal areas where Jazz needs to interact with the existing code base or its output [46]. First, inserting code as instrumentation means that Jazz must interact at a low level with the JIT compiler during code generation. Second, the generated instrumentation code needs data to direct how it operates and space to store its results. Thus, memory allocation must be considered. Third, each test planner requires some facilities that may already exist in a compiler, such as control or data flow analysis. Finally, Jazz must efficiently parse and interpret the test specification language to specify what and where to test.

3.3.1 Implementing the Test Driver

The implementation of the `FrameworkHarness` class contains several static member functions that are called from the base RVM code. These methods are hooks that were inserted into parts of the existing RVM code base (such as changes to the commandline argument parser and callbacks in the JIT compiler) to interact with the RVM in the four places identified above. `FrameworkHarness` does test selection and invokes the appropriate test implementations to instrument the code. Additionally, `FrameworkHarness` registers callbacks for the RVM's exit notification to report final coverage results (as the RVM is shutting down). Furthermore, the class is also a place to store global settings, such as a verbose output flag to dump detailed messages for tracing and debugging.

The RVM is written in Java and runs itself through its own compiler. To make this

Table 3.1: Interface for adding instrumentation in a method-oriented JIT environment.

Callback function	Description
`onCompilation(Method m)`	A method is about to undergo compilation.
`onBytecode(Bytecode b, int i)`	The i^{th} bytecode is about to have machine code generated for it.
`afterBytecode(Bytecode b, int i)`	The i^{th} bytecode has just had machine code generated for it.
`afterCompilation(Method m)`	A method has been completely compiled but is not yet being executed.

possible, it uses a bootloader with a minimally compiled bootimage that enables the RVM to begin. Since calls to `FrameworkHarness` are compiled into methods that are part of the bootimage, `FrameworkHarness` must be added to the bootimage as well. It is the only class in Jazz that appears in the bootimage. Jazz takes special precaution to avoid instrumenting internal RVM code since the bootstrapping process is delicate and much of the Java class library is unavailable.

3.3.2 Instrumentation & Code Generation

Access to the compiled method code is required to instrument it. To support this, there needs to be a low-level interface between the JIT compiler and Jazz. There are four points during compilation where control might need to intercepted and to have Jazz or one of its test implementations do work. These four events and the interface for capturing this interaction are shown in Table 3.1. The RVM's JIT compiler is modified to call these methods for a test implementation that registers a handler for the events.

The earliest event handler necessary is for when a new method is about to undergo compilation. For node and branch coverage, the `onCompilation` event is used for initialization. This is the time to construct object instances, register exit callback handlers, and do the work of determining where to insert instrumentation during the upcoming compilation phase.

The two bytecode-oriented methods are used to insert static instrumentation directly into the generated machine code stream. The static test techniques in Jazz's current test library implement the `onBytecode` event to insert probes at the beginning of a basic block. However, none of the tests in Jazz's library make use of the `afterBytecode` event. It is provided for completeness should a new test require it. The `afterCompilation` event is a natural place to put clean-up for static instrumentation.

3.3.3 Memory allocation

In general, four types of memory are used in Jazz:

1. Object instance memory for the Java code that interfaces with the RVM;
2. Executable memory for instrumentation;
3. Storage for recording coverage and directing runtime instrumentation (e.g., conditions for removal of instrumentation for demand-driven testing); and,
4. Method local storage to maintain test state.

A significant consequence of the RVM's design is that the Java code and data that comprise the RVM can and do share the same memory regions as the applications that run on top of it. This structure means that as the RVM is extended to support Jazz, any of the Java code that implements a test in the test library can increase pressure on the garbage collector (GC), possibly degrading performance with more collection rounds. Despite this potential drawback, the use of heap memory is vital to conveniently express a test implementation in Java itself. For Jazz, the heap is used for any operation done during JIT compilation, except storing instrumentation code and data used and generated by the instrumentation code as it executes.

Jazz avoids Java's memory for instrumentation as it can adversely interact with the garbage collector; if the JVM uses a copying GC, the instrumentation might move during execution. Rather than implement an additional level of indirection (as a JVM might), Jazz avoids the extra lookup costs and allocates space that is outside of the Java heap.

Most of the test techniques implemented in Jazz share some common code between instrumentation probes to reduce code footprint. The shared code is the test *payload code*.

It performs test actions and records coverage results. The RVM has special facilities for calling low-level system calls (e.g., `mmap`) to allocate its own heaps. This functionality is reused to allocate an executable page of memory to store the test payload code.

The test payload is parameterized with a *test plan* that drives and/or records the results of testing. The test plan is a data argument passed to the payload. For instance, the test plan lists the predecessors in Static Branch so the appropriate counter can be set as covered at runtime. The memory for a test plan should not move during execution, and Jazz allocates it via `mmap`. (Instead, `malloc` could be used as the region does not need to be executable—it is data.)

The final piece of memory needed for Jazz involves coverage tests that require persistent state, such as branch coverage. This test needs state to be propagated between the execution of two probes. With Jazz's approach of placing instrumentation in basic blocks, to record an edge the probe at an edge sink needs to know which basic block proceeded it during execution (i.e., the source), and thus, which edge to mark as covered. This state is local storage, as each separate activation of a method needs its own copy of the state. For branch coverage, each probe stores a block-unique identifier. The static implementation of branch coverage uses the address of the previous block. The demand implementation uses the address of the previous probe's test plan. Each address requires one word of storage and so my implementation places these into the activation record of the JIT-ed method.

To support memory storage in Jazz requires modifications to the RVM which have grown increasingly more difficult in each new release. A perhaps simpler alternative would be to have a separate manually-managed stack for storing test-specific state. This is the model Jazz will move to in the future as it provides more of the isolation from RVM changes and eases continued development.

An interesting implication of memory allocation outside the JVM is that pointers and low-level operations for initialization and reading memory must be used. The RVM has a special facility for providing this capability for its own use called Magic [25]. Magic are special snippets of what appear to be Java methods and code that are intercepted by the JIT and replaced with operations that would be normally prohibited in Java code.

3.3.4 Implementing Test Specification

A test specification, written in testspec, can be passed to Jazz through a file specified on the command line. Alternatively, a test specification, if simple enough, can be given directly as a command line argument (option `-I`). The `FrameworkHarness` consumes the specification to drive testing. Internally, it creates an intermediate representation (IR) of the test specification.

Each rule in a testspec specification is parsed to construct a "matching" object. A matching object is a representation of a rule. This object includes a `matches` method that can be used to determine whether the rule should be applied to a particular method. My first implementation of testspec's IR arranged the matching objects in a list. On each method load, the list was linearly searched to check for a matching rule. However, for complex test specifications, this approach proved to be computationally expensive, slowing down runtime performance of the Java program under test. For example, the *javac* program in SPECjvm98 has 742 methods. A linear search for a matching test is responsible for 4% of total overhead.

The intermediate representation was reworked to use a hashtable to arrange the matching objects. The hashtable is checked when a method is loaded for a corresponding matching object. There were two complications. The first was that wildcards do not make sense to hash. Instead, literal (class name, method name) pairs are placed in a hashtable and patterns are left in an array to be searched. The second issue was that neither `HashMap` nor `Hashtable` were in the RVM's bootimage. Thankfully, the RVM developers provide an internal hashtable class that Jazz reuses rather than writing new one.

In general, it may be useful to have full regular expression support. However, this has proven unnecessary in practice. The simple pattern matching scheme used in testspec is sufficient. Java has native support for regular expressions in class libraries that may be able to be reused, but those classes are not part of the bootimage.

3.3.5 JVM Support for Software Testing

In developing the support services and test library for Jazz, there are several places where a JVM developer can ease implementation of a tool like Jazz.

A foundation of structural testing is to discover control flow properties of a region of code. To this end, at bare minimum, a control flow graph (CFG) is necessary. The RVM provides a CFG generator that scans the Java bytecode and determines the basic blocks and their predecessors. Jazz needs both predecessor and successor information. The CFG is augmented with this information and other information that is useful about the structure of the code.

Jazz needed several analyses such as pre- and post-dominator information to support the structural testing optimizations suggested by Agrawal. The RVM already has facilities for these algorithms in its optimizing compiler. However, the code is tied closely to the optimizer. It was too cumbersome to reuse this code and so the necessary analyses were written from scratch. The implementation of Jazz would have been simpler with support for control and data flow analyses that is separated from the optimizer (e.g., such as Phoenix provides [9]).

Finally, a JVM could provide a rich and varied set of events to register callbacks. The RVM provides several callbacks, many of which Jazz does not need, but the exit handler callbacks were useful for reporting the collected coverage results. For instance, the code generation interface of Section 3.3.2 would be convenient to achieve as much isolation from the JVM codebase as is possible.

3.4 STATIC BRANCH AND NODE TESTING WITH JAZZ

Figure 3.3 shows how static testing operates. A testspec specification is first loaded and parsed. When a method is about to be compiled, an `onCompilation` event is sent to the `FrameworkHarness`. If the current method matches a rule for a static test, Jazz instantiates a static test planner. The planner is responsible for instrumentation code generation,

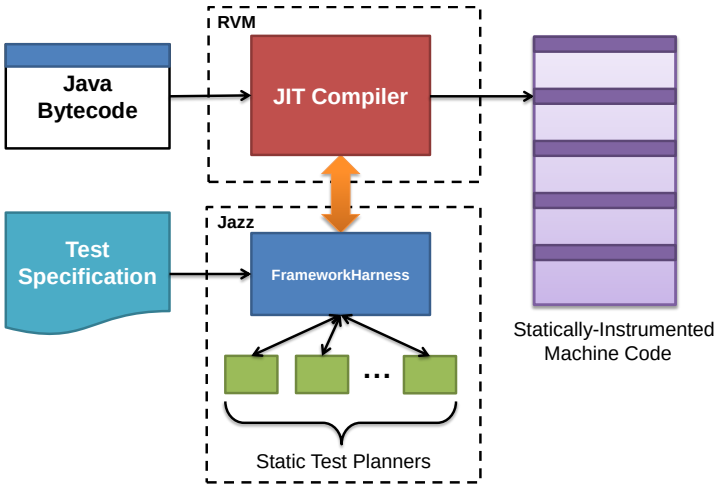

Figure 3.3: Static instrumentation in Jazz.

test plan generation, and result recording. After the planner is initialized, the JIT compiler continues and emits the method prologue and enters its main translation loop. In a non-optimizing compiler, one bytecode expands into one or more machine instructions. For static testing, `FrameworkHarness` intercepts control via the `onBytecode` event and transfers control to the test planner to insert instrumentation prior to the bytecode being compiled.

3.4.1 Static Node Testing

Figure 3.4 shows an example of how the `onBytecode` event is used in the Static Node test planner. For all static tests, there are three common elements:

1. a *seed set* of code locations where the planner will sew permanent instrumentation probes (to possibly invoke the test payload);
2. a *test payload* that implements the work of recording the desired coverage information; and,
3. a *test plan* that provides storage for recording coverage.

```
1 public void onCompilation(NormalMethod m) {
2    cfg = CFGBuilder.build(m);
3    seedSet = cfg;
4 }
5
6 public void onBytecode(
7       int bytecodeAddress,
8       Assembler asm) {
9
10    int index = seedSet.indexOf(
11       new Integer(bytecodeAddress));
12
13    if(index < 0) {
14       return;
15    }
16
17    asm.emitPUSH_Reg(GPR.EAX);
18    asm.emitMOV_Reg_Imm(GPR.EAX, plan[index]);
19    asm.emitMOV_RegInd_Imm(GPR.EAX, 1);
20    asm.emitPOP_Reg(GPR.EAX);
21 }
```

Figure 3.4: The static node coverage test planner.

```
1 public void onCompilation(NormalMethod m) {
2    cfg = CFGBuilder.build(m);
3    seedSet = Agrawal.getProbeSet(cfg);
4 }
```

Figure 3.5: Extending the Static Node planner to incorporate H. Agrawal's algorithm.

The seed set and test plan are the implementations of the same items from the formalization of Section 3.1.

The seed set is generated as part of the `onCompilation` event handler method (lines 1–6). Static instrumentation is inserted at the start of a basic block to avoid rewriting control flow transfers. A control flow graph that is annotated with each basic block's starting bytecode address is built. These addresses become the locations where permanent instrumentation is inserted, i.e., the seed set.

In the `onBytecode` event handler method of the static test planner, a check is done to determine whether the current bytecode is a seed. If so, the RVM's assembler is used to emit instrumentation code. For node coverage, the instrumentation is simple enough that it can be inserted entirely inline. For more complex tests, Jazz supports instrumentation probes that call an out-of-line function for a shared test payload. This probe type pushes location-specific information onto the stack to pass information to the payload.

Agrawal's algorithm reduces the number of instrumentation probes needed to record complete coverage using control flow analysis on a method under test. The algorithm is incorporated as a support service since it can be used by more than one test technique. With this service and the basic Static Node, I created a new test, Static Node Agrawal, that does node coverage with reduced instrumentation. I extended the basic Static Node planner to use the algorithm as is shown in Figure 3.5. The `onCompilation` event handler is slightly adjusted to call the Agrawal support service. Based on the minimized instrumentation points, a new seed set is generated. No other changes to the basic Static Node test were required.

3.4.2 Static Branch Testing

Static Branch coverage testing is implemented in a similar fashion to Static Node. The `onCompilation` event is handled identically, as a control flow graph still needs to be built. The seed set is once again all basic blocks in the method. The main difference between Static Node and Static Branch involves the `onBytecode` event. Static Node recorded coverage directly as shown on line 21 of Figure 3.4. Static Branch replaces this line with a call to the payload shown in Figure 3.6. Lines 1–2 load the identifier of the block that immediately preceded the current so the edge can be marked as hit in lines 9–13. This block sets that same state for the next probe encountered in line 3.

Another difference from Static Node involves the test payload. Static Branch uses an out-of-line function to invoke the test payload. The payload is relatively large—it takes 14 instructions to determine which control flow edge was taken at runtime. If this code was fully inlined, the generated machine code becomes very large, which puts unwanted pressure on the instruction cache. Instead, the code contains a call to the shared payload functionality.

The test plan for Static Branch Coverage is implemented as a table in memory. The layout is shown in Figure 3.7. The table contains a row for each instrumentation probe that is in the test region. The first entry in a row is the number of control flow predecessors (and therefore is the number of incoming edges). For each predecessor, there are two additional entries in the table. The first is the address of that predecessor. The address is matched at runtime in the test payload to the value in the previously hit block variable to determine the control flow edge that was taken to the current probe. When the appropriate value is found, the associated coverage counter is set to one to indicate the edge was covered.

Similar to Static Node, H. Agrawal's algorithm can be used to reduce the amount of instrumentation for branch coverage testing. The basic Static Branch test planner was extended to apply H. Agrawal's algorithm in the same way as was done for Static Node.

```
1        mov ebp, dword ptr [esi + fpOffset]
2        mov ebx, dword ptr [ebp - 8]
3        mov dword ptr [ebp - 8], edx
4        mov ecx, dword ptr[edx]
5        test ecx, ecx
6        jz EXIT
7
8        add edx, 4
9  L1:   cmp ebx, dword ptr [edx]
10       jne NEXT
11
12       mov dword ptr [edx + 4], 1
13       jmp EXIT
14
15 NEXT: add edx, 8
16       loop L1
17
18 EXIT: ret
```

Figure 3.6: Static payload for branch coverage testing.

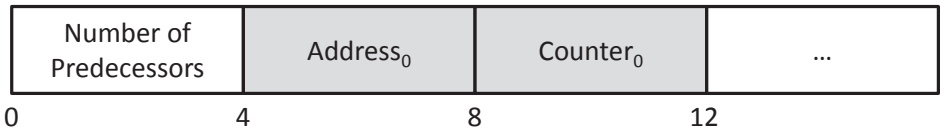

Figure 3.7: A row of the test plan for Static Branch as laid out in memory.

3.5 SUMMARY & CONCLUSIONS

In this chapter, I describe a two-part framework for performing branch and node testing. The first part was a formal framework for describing a structural test technique that encompasses instrumentation probe location, runtime actions, conditions for those actions to occur, and state.

Additionally, I developed Jazz, a flexible and extensible practical framework for realizing the implementation of structural tests. Jazz provides a set of support services to perform code generation, memory management, control flow analysis, and handle interaction with a JVM. In addition to these services I built a library of test planners to do branch and node coverage testing.

New test planners can be created by extending existing tests from the library or by using the support services directly. Jazz also provides a test specification language, *testspec*, that allows multiple test planners to be combined and applied in a single run of the application being tested.

Jazz is implemented on top of the Jikes RVM for Java. For Jazz's test library, I have described and implemented Static Node, Static Agrawal Node, Static Branch, and Static Agrawal Branch. Both parts of the framework greatly eased the development of these structural tests.

4.0 DEMAND-DRIVEN STRUCTURAL TESTING

TRADITIONAL TECHNIQUES for recording structural test coverage rely on statically-inserted instrumentation probes that remain throughout the lifetime of the program. Coverage, however, is a boolean property that only needs to be recorded on the first execution of a node or branch in a program. When there is a loop in the test region, many of the dynamic executions of those instrumentation probes occur after any coverage information has been recorded. These extraneous executions incur runtime overhead in terms of both CPU time and memory without contributing new coverage data.

It seems that a better course of action would be to remove instrumentation when it is no longer necessary. However, removal will also incur runtime cost and so the question becomes: "When is it appropriate to remove instrumentation probes that are no longer necessary?"

There is a large spectrum of choices for the appropriate removal time ranging from "never" with traditional static instrumentation to the immediate removal of a probe after its execution. Other choices include using a time- or event-driven collection scheme to periodically remove instrumentation. Tikir and Hollingsworth use a time-periodic removal scheme to collect away instrumentation probes for node coverage on C programs. However, the runtime overhead they report is high, up to 3.26 times slower than the uninstrumented execution [67].

In light of the poor performance of never removing probes and time-periodic removal, this chapter explores the other extreme: instantaneous removal of instrumentation.

Instantaneous removal of instrumentation from an instruction stream requires self-modifying code. Memory hierarchies make many assumptions about the read-only nature of code in order to maintain consistency and to cache it. This means that removing a

single instrumentation probe is likely to incur a significant overhead and reduce the gain provided by not using the cheaper and cache-friendly static instrumentation probes. For a technique that relies on self-modifying code to improve upon static's permanent instrumentation probes, sufficient dynamic executions of instrumentation probes must be eliminated to amortize this removal cost.

Demand-driven Structural Testing (DDST) uses the program's own execution to drive where instrumentation is inserted and removed. Demand-driven removal attempts to alleviate the problem of permanent instrumentation by removing probes when they are not longer necessary. Demand-driven insertion allows for the execution of the program to place probes only along the frontier of execution, preventing probes from being placed into infeasible paths or into code that is not being tested in this run.

Both branch and node coverage can benefit from the demand-driven approach. However, with the runtime management of probes that DDST implies, care must be taken in planning the actions of the instrumentation code itself. DDST is implemented on top of Jazz and uses the same model of planning as was used for Static and Static Agrawal testing. This chapter describes the challenges and approaches for designing and implementing DDST in Jazz.

4.1 PLANNING FOR DDST

The main function of the test planner is to determine where and how to test a code region. Using the specification and the intermediate code for a test region, the test planner determines the actions necessary to carry out tests. These actions are the run-time activities that collect coverage information and instrument the test region. The actions form the basis for the test plan. In the next sections, I will discuss some of the test planner challenges and implementation strategies for DDST.

To generate a test plan, a planner needs to determine when to insert probes, where to instrument a test code region, and what to do at a probe. There are three cases the planner has to consider when deciding when to insert and delete instrumentation. First, it

must identify which probes are *seeds*. Seeds are those probes that are initially inserted in a test region. Second, it needs to determine which probes are used for coverage and can be inserted and removed on-demand along a path. Finally, the planner has to determine the lifetime of a probe and whether it must be re-inserted after being hit by its "control flow successor" basic blocks (*anchors*).

The test planner also must identify the locations of probes in a test region. These locations correspond to seed, coverage, and anchor probes. Seed locations must be marked in a table, called the probe location table (PLT), that drives the dynamic instrumenter, which inserts the seed probes. Coverage and anchor locations also have entries in the PLT to hold information needed by the probes. Coverage locations have an entry in the PLT to record coverage information.

The last task of the planner is to determine what actions should take place at a probe. In some cases, different payloads or combinations of payloads may be used at different probes and the planner needs to select the appropriate payload.

4.1.1 Planner Actions

Actions in a test plan are implemented with a test probe and payload. Probes can be inserted in a code region at any basic block where test actions need to take place. A test plan may have multiple payloads, which can be invoked by different probes, and multiple probes may be inserted at the same location to call different payloads. The test plan uses the PLT to encode probes and their locations. A PLT entry has a probe type, a payload, and a list of probes to insert (and in some cases, to remove). Additional fields can be added to the PLT by the planner. The test plan also has data storage, including global memory that is persistent with program scope (i.e., there is a single global storage area) and local storage with method scope. Global storage is used to hold test results for multiple test runs (i.e., what has been covered) and the local storage is used to hold temporary values needed by a payload. Other storage scopes can also be incorporated into a plan (e.g., thread or class scope).

4.1.2 Node Coverage Planner

With the formal notation framework defined in Section 3.1, the test plans for node coverage with DDST can be expressed. An addition for DDST is necessary: There needs to be a set of locations, L, in which to place probes at runtime.

- The seed set S contains the entry point(s) of the test region.
- A set of $|N|$ test plans, P, where P_i is a test plan corresponding to a probe in basic block i and contains:
  - A set $L \subseteq N$ of locations to place instrumentation, and are those nodes that are CFG successors of node i.
  - A set C of boolean flags (`Cov`), all initially set to **false**, where `Cov[j]` records if node j has been covered.
  - A set A of actions, where the actions for P_i are:
    1. Record the coverage of node i in `Cov[i]` = **true**.
    2. For each location L_j, if `Cov[j]` = **false**, place a probe at L_j.
- The global state M is the union of all sets C, $|M| = |N|$.

The test planner for node coverage must populate S, allocate the space for M, and then create a set of test plans, P, for each node in the region to test, filling in L with the appropriate code locations. The space for M and L is allocated in the PLT, which is the implementation of the set of test plans, P.

The planner that constructs these test plans for node coverage is shown in Algorithm 4.1. Each node in the region to be tested, `testRegion`, is added to the PLT with the node coverage payload. If there are no predecessors of the node in the region to be tested, lines 6–8 mark the block as a seed. Seed probes then place their control flow successors in a demand-driven fashion.

4.1.3 Branch Coverage Planner

The branch coverage test planner determines where to place instrumentation probes in a region to ensure that all edges can be marked as covered when they are traversed.

```
1: CFG G ← buildCFG(testRegion)
2: for all Block b ∈ G.nodes do
3:   PLT[b].payload ← nodePayload
4:   PLT[b].insert ← b.successors
5:   // Check if block b is an entry block (a seed)
6:   if b.predecessors ⊄ G.nodes then
7:     PLT[b].seed ← true
8:   end if
9: end for
```

Algorithm 4.1: The node coverage planner.

For branch coverage, the seed blocks are the entry points into a test region. These seed blocks insert instrumentation when control passes through an entry. Seeds are once again identified as basic blocks that have one or more predecessors outside of the test region.

A more difficult issue is how to record which edges are executed, and when probes need to be inserted and removed. To cover an edge, two probes are executed: one in the edge's source basic block and one in the sink block. The probe in the source records the beginning of an edge and the probe in the sink marks the edge as covered. This information is recorded as part of thread-local state in a variable that records the previously hit block. The branch coverage planner needs to determine which instrumentation probes to insert and delete when a block is hit to ensure that the correct edge is recorded.

This yields a similar description of branch coverage test planning in the notation of Section 3.1 as for node coverage:

- The seed set S contains the entry point(s) of the test region.
- A set of $|N|$ test plans, P, where P_i is a test plan corresponding to a probe in basic block i and contains:
  - A set $L \subseteq N$ of locations to place instrumentation, and are those nodes that are CFG successors of i.
  - A set C of boolean flags (`Cov`), initially set to **false**, where `Cov[i→j]` records if the edge between i and the probe at L_j has been covered.

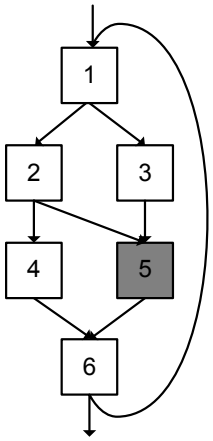

Figure 4.1: The shaded basic block is a stranded block.

  - A set A of actions, where the actions for P_i are:
    1. Look up the previously hit block, b.
    2. Record the coverage of edge $b \rightarrow i$ in `Cov[b→i]` = **true**.
    3. For each location L_j, if `Cov[i→j]` = **false**, place a probe at L_j.
    4. Set the previously hit block, $b \leftarrow i$.
- The global state M is the union of all sets C, $|M| = |E|$ plus an additional element of storage for the previously hit block.

However, this ends up being insufficient for DDST branch coverage to yield correct results.

4.1.3.1 Stranded Blocks DDST records coverage using a pair of probes at the source and sink basic blocks of a control flow edge. This is convenient for the ease of inserting and removing probes dynamically as branches and fall-throughs do not need to be rewritten in the machine code. However, it can lead to a situation in which one or both of the probes has been removed via a previous execution of the region being tested, resulting in an edge which cannot be recorded as covered.

Figure 4.1 shows a CFG with this problem. If the path $1 \rightarrow 2 \rightarrow 4 \rightarrow 6$ is taken and then $1 \rightarrow 3 \rightarrow 5 \rightarrow 6$ is taken, there is no instrumentation in the region to record when

the edge $2 \rightarrow 5$ is covered. This problem is the *stranded block* problem and block 5 is designated as the stranded block. The problem exists because there are multiple paths into block 5 and at least one path that excludes block 5 but includes a control flow predecessor (block 2), allowing for that probe to be removed before all of the outgoing edges are recorded.

A stranded block b can be identified by defining three predicates:

Fork(x) that returns true if x has multiple control flow successors.

Merge(x) that true if x has multiple control flow predecessors.

Predecessor(x,y) that returns true if x is a control flow predecessor of y.

Thus, $Stranded(b)$ can be defined as:

$$Stranded(b) = Merge(b) \wedge \exists x(Predecessor(x, b) \wedge Fork(x))$$

The planner for demand-driven branch testing must identify any stranded blocks in a test region and must keep some instrumentation probes in the program until the stranded block problem is resolved. An approach to handling a stranded block is to leave instrumentation in its predecessors until the stranded block's incoming edges are all covered. In the example CFG of Figure 4.1, instrumentation must remain in blocks 2 and 3 until the edges $2 \rightarrow 5$ and $3 \rightarrow 5$ are covered. Care must be taken to ensure that an instrumentation probe remains in block 2 even if the edge $2 \rightarrow 4$ is taken. Thus, the solution requires more than just adjusting the actions taken by the probe in the stranded block itself, it also affects any "siblings" of the stranded block.

By the definition of a stranded block, the common If-Then structure as shown in Figure 4.2 also has a block that is considered to be stranded. Block 3 has multiple predecessors and one of them, block 1, has multiple successors. If the path $1 \rightarrow 2 \rightarrow 3$ is taken, there will be no probes left to record the path $1 \rightarrow 3$. The solution presented above applies to If-Then stranded blocks as well.

However, another issue arises in the opposite situation. If the path $1 \rightarrow 3$ is taken first, there remains a probe in block 2 that has no prior probe to record the source of the edge. Since there is only one possible edge into block 2, a simple solution is to use a

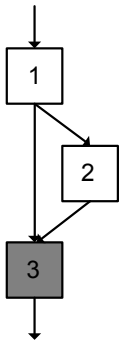

Figure 4.2: The shaded basic block is also a stranded block.

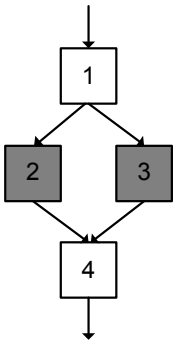

Figure 4.3: The shaded basic blocks are singleton blocks.

```
1: CFG G ← buildCFG(testRegion)
2: for all Block b ∈ G.nodes do
3:   PLT[b].type ← regular
4:   // Check if block b is an entry block (a seed)
5:   if b.predecessors ⊄ G.nodes then
6:     PLT[b].seed ← true
7:     // Check if block b is a singleton
8:   else if |b.predecessors| = 1 then
9:     PLT[b].type ← singleton
10:  end if
11:  // Check if block b is stranded
12:  for all Block p ∈ b.predecessors do
13:    if |p.successors| > 1 ∧ |b.predecessors| > 1 then
14:      PLT[b].type ← stranded
15:    end if
16:  end for
17: end for
```

Algorithm 4.2: First phase of the branch coverage planner.

probe that does not need to look up the previously hit block but instead automatically records the appropriate incoming edge's coverage. This type of block is called a *singleton*. Singleton blocks are orthogonal to stranded blocks, as is shown in Figure 4.3. While this If-Then-Else CFG does not have a stranded block according to the definition, if either path is taken, the opposite path will be left without a probe to record the source. Thus, blocks 2 and 3 are both designated singletons and will automatically record the incoming edge without looking in the previously hit block variable.

To handle coverage of normal blocks, stranded blocks, and singleton blocks, the planner for demand-driven branch coverage is implemented in two phases. The algorithm for the first phase of the planner is shown in Algorithm 4.2. The planner creates a CFG for the test region on line 1. Next, it iterates over basic blocks to determine whether they are a seed, singleton, stranded, or regular block. Initially, on line 3, a block is treated as a regular block that inserts probes in its successors blocks. Lines 5 and 6 check whether the block has any predecessors that are not in the test region and sets the PLT field `seed`

```
1: for all Block b ∈ G.nodes do
2:   for all Block s ∈ b.successors do
3:     s.addAction(COVERAGE_SUCCESSOR, GLOBAL[b→s])
4:   end for
5:   if b.type = stranded then
6:     for all Block p ∈ b.predecessors do
7:       p.addAction(ANCHOR, GLOBAL[b])
8:       for all Block s ∈ p.successors do
9:         s.addAction(ANCHOR, GLOBAL[b])
10:        if s ≠ b then
11:          p.addAction(ANCHOR, GLOBAL[b])
12:        end if
13:      end for
14:    end for
15:  end if
16: end for
```

Algorithm 4.3: Second phase of the branch DDST planner.

to true, if so. When a block has a single predecessor and it is not a seed, then it is a singleton, as shown on lines 8–10. In this case, the singleton's predecessor is recorded in a table in global memory. At run-time, when the singleton payload is invoked, it accesses the table to get its predecessor. Finally, lines 12 to 16 check for stranded basic blocks.

Once the various types of blocks have been determined, the second phase of the planner begins. The steps are shown in Algorithm 4.3. This phase walks over the basic blocks in the test region again, examining the type of block as determined in the previous phase. For regular basic blocks, line 3 adds an action to the block to dynamically place each control flow successor predicated on the coverage of the edge from the block to its successor. Lines 5–16 handle a stranded block. A boolean variable is allocated in the global state representing the stranded block. For each basic block that is a predecessor of the stranded block, the instrumentation probe in that predecessor block needs to be replaced when the stranded block is hit (lines 6 and 7). Those same probes must also be replaced if a path is taken through a sibling of the stranded block. This situation is handled on lines 8–13. These actions are deemed to be *anchors* as opposed to probe

```
16: // Is this the special case of the If-Then stranded block?
17: if PLT[b].type = stranded then
18:   if |b.predecessors | = 2 then
19:     if one predecessor has the other as a successor then
20:       PLT[b].type ← ifthenstranded
21:       root ← predecessor of b that has 2 successors
22:     end if
23:   end if
24: end if
```

Algorithm 4.4: Handling a If-Then Stranded Block in Phase 1.

placements that are meant to record coverage.

4.1.3.2 Improving Stranded Block Performance The general solution to the stranded block problem requires many probes to remain as anchors even after they have recorded the coverage of all incoming edges. If-Then stranded blocks can be dealt with in a simpler way than the general solution, which provides an opportunity to improve the performance of the general case.

The realization that enables a different solution for an If-Then stranded block than the general case is that the singleton block allows for only replacing the stranded block's predecessors until the edge $1 \rightarrow 3$ is taken. The general solution would leave probes in until both $1 \rightarrow 3$ and $2 \rightarrow 3$ are covered, but in essence, a singleton block enables the reseeding of a region that has had instrumentation probes removed by previous executions.

To implement this improved case, Algorithm 4.2 is augmented to identify If-Then stranded blocks. Algorithm 4.4 shows the logic that is inserted at line 16 of Algorithm 4.2. It identifies an If-Then stranded block by its control flow structure and sets a new type.

Algorithm 4.5 is the necessary addition to the second phase of the branch coverage planner, as was shown in Algorithm 4.3. The code is inserted before the original line 5, and the stranded condition becomes an else-if. The new line 7 contains the key to the If-Then

```
5: if b.type = ifthenstranded then
6:   for all Block p ∈ b.predecessors do
7:     p.addAction(ANCHOR, GLOBAL[root→b])
8:   end for
9: end if
```

Algorithm 4.5: Handling a If-Then Stranded Block in Phase 2.

stranded block improvement: The predicate on which the stranded block's predecessors are replaced is reduced from the special stranded block global state in the general case to being predicated on the edge from the if-condition block (the root) to the stranded block. The singleton that is the body of the if statement reseeds the region, allowing for the edge from it to the stranded block to be covered even if the instrumentation probe has been removed from the root.

4.1.4 Pre-seeding

Another improvement to the runtime performance of both branch and node testing is to redefine the seed set of a region to be tested. Seeding only the entry points in the region allows for demand-driven insertion, where instrumentation probes are placed along the frontier of execution, that is, in basic blocks that control flow could immediately reach. This leaves infeasible or rarely-executed code paths mostly instrumentation free.

However, due to instruction cache behavior and the costs of self-modifying code, demand-driven insertions could be a source of runtime cost that a planner for DDST can mitigate. If every block in a tested region is considered a seed, the probes can be placed after the JIT-compiler has generated the code for the method but before that code actually executes.

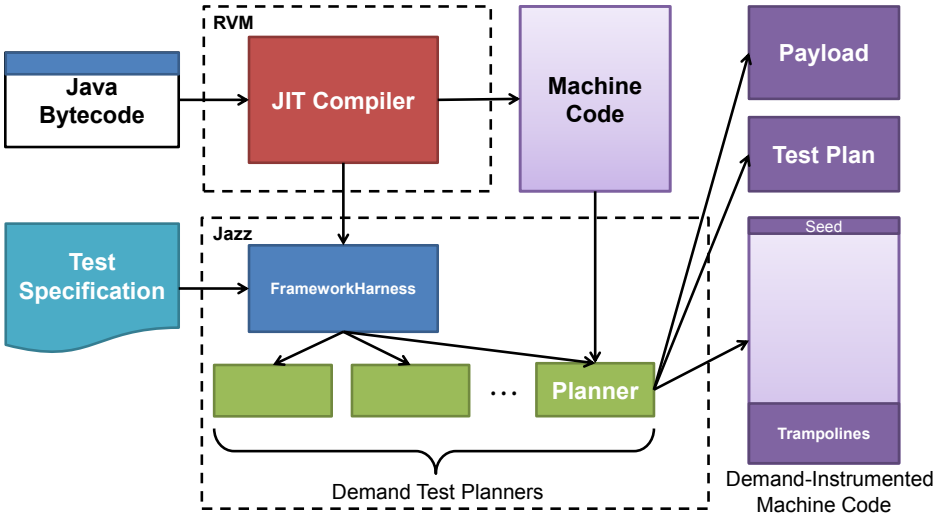

Figure 4.4: Demand-driven instrumentation in Jazz.

4.2 IMPLEMENTING DDST IN JAZZ

Demand-driven testing adds instrumentation after compilation of a method. Figure 4.4 shows an overview of DDST under Jazz. Jazz's `afterCompilation` event is handled as part of `FrameworkHarness`. This handler checks whether the current method (just compiled) matches a rule that applies a demand-driven test. If so, it invokes the `instrument` method of the appropriate demand-driven test planner.

Figure 4.5 shows the base class for demand-driven test planners in Jazz. Subclasses perform work via the `instrument` method which is invoked from `FrameworkHarness`. The arguments to `instrument` capture the JIT-ed machine code as well as how that code maps to the original bytecode. The `insertFastBreakpoint` method exposes the foundation all demand-driven structural tests share: the dynamic insertion and removal of instrumentation via fast breakpoints.

Demand testing extends the three common elements from static testing and adds a fourth. All demand tests have in common:

1. a *seed set* of instrumentation probes that are inserted before the method executes;

```
1 public abstract class StructuralTest {
2    public abstract void printResults(
3          PrintWriter out);
4
5    public abstract MachineCode instrument(
6          Assembler asm,
7          NormalMethod method,
8          int[] bytecodeMap);
9
10   protected void insertFastBreakpoint(
11         ArchitectureSpecific.CodeArray instrs,
12         int nInsertAddr,
13         int nDestAddr){
14
15      int nImm = nDestAddr - (nInsertAddr + 5);
16
17      //JMP rel32 (relative to next instruction)
18      instrs.set( nInsertAddr+0, (byte) 0xE9);
19      instrs.set( nInsertAddr+1, (byte)((nImm >> 0) & 0xFF));
20      instrs.set( nInsertAddr+2, (byte)((nImm >> 8) & 0xFF));
21      instrs.set( nInsertAddr+3, (byte)((nImm >> 16) & 0xFF));
22      instrs.set( nInsertAddr+4, (byte)((nImm >> 24) & 0xFF));
23   }
24 }
```

Figure 4.5: The base class for all demand-driven structural tests.

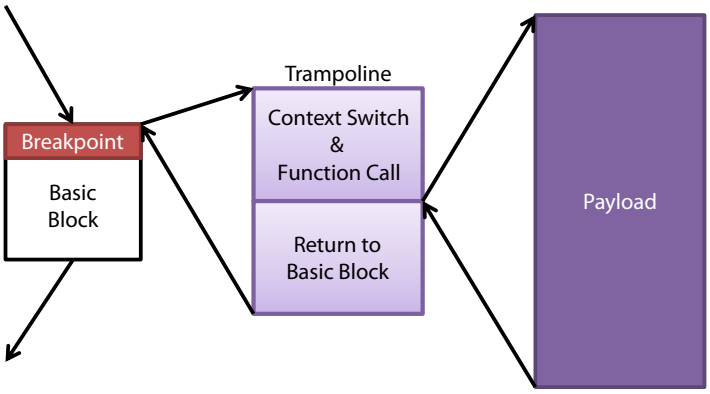

Figure 4.6: Breakpoint implementation on x86.

2. a *test payload* to record coverage information;
3. a *trampoline* targeted by a fast breakpoint that sets up location-specific parameters for the instrumentation probe's call to the payload; and,
4. a *test plan* that contains directions for each instrumentation probe in terms of what other probes to place and to remove, and provides storage for recording coverage information.

The Demand Node test planner is similar to the Static Node planner. It constructs a CFG for a method and determines a seed set of locations to place fast breakpoints. However, instead of inserting the instrumentation inline, the test planner overwrites instructions in a basic block with a control transfer (i.e., fast breakpoint) to an associated trampoline. The trampolines set up the function call to the shared payload. The trampolines are emitted at the end of the machine code array. These components are shown on the right side of Figure 4.4.

4.2.1 Dynamic Instrumentation for the x86

DDST under Jazz uses fast breakpoints [40] to implement instrumentation probes as shown in Figure 4.6. A fast breakpoint replaces an instruction in the target machine code with a jump to a trampoline. The breakpoint handler calls the test payload and it

executes the original instruction that was replaced by the jump. Fast breakpoints have low overhead and can be easily inserted and removed on binary code. When implementing fast breakpoints there are essentially two choices. The first choice is to execute the original instruction as part of the breakpoint handler. The second choice copies the instruction back to its original location where it is executed when the breakpoint handler completes. Hence, these breakpoints are "transient" and similar to the invisible breakpoints used by debuggers to transparently track program values and paths.

A consequence of transient breakpoints is probes do not remain in a test region once executed. If a permanent probe is needed, then the test planner has to re-insert the probe. Re-insertion can be done by placing probes in the successors (anchors) to a block that needs a permanent probe. The successor probes re-insert the original probe when executed and remove themselves and their siblings. While fast breakpoints can be implemented to make them permanent, variable length instruction sets, like x86, complicate the implementation. Instead, transient breakpoints simplify and increase the portability of the instrumentation interfaces.

On the x86, copying the instrumented instruction back to its original location works better than executing the instruction in the handler. If the instrumented instruction is executed in the handler, then instructions have to be decoded to find instruction boundaries because an entire instruction must be copied to the handler. Indeed, in some cases, multiple instructions may have to be copied and executed in the handler because the breakpoint jump can span several instructions. The breakpoints do not know anything about the instructions where a breakpoint is inserted, which significantly simplified their implementation. The trade-off is for a breakpoint to remain, it must be re-inserted after the original instruction is executed.

4.2.1.1 Short Blocks A second issue arises when using fast breakpoints on a variable-length instruction set such as the x86's. If the fast breakpoint is implemented in terms of a multi-byte opcode, it is possible that the fast breakpoint will overwrite more than a basic block in the original instruction stream. This is the *short block problem*. If the fast breakpoint extends past the end of a short block, there may be a control flow transfer

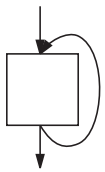

Figure 4.7: A reflexive basic block.

around the instrumented basic block that jumps into the middle of the fast breakpoint instruction, resulting in an invalid opcode being executed.

There are several possible solutions to the short block problem. This problem is not unique to structural testing but also occurs in traditional debugging, and so the architecture supports a single byte breakpoint trap (`int 3` encoded as `0xCC`) that guarantees it will never extend beyond a single basic block. However, traps and the resulting signals involve context switches into and out of the operating system, incurring significant delays. Another approach is to use a smaller opcode for jumping, usually with a smaller relative offset immediate. A full 32-bit relative jump requires five bytes of encoding on the x86, but a 16-bit jump can be encoded in 4-bytes (1-byte opcode, 2-byte immediate, and 1-byte 16-bit mode prefix). An 8-bit jump is unlikely to be sufficient to reach a trampoline, but might be appropriate to use in some situations.

A final approach is to simply identify short blocks while doing the JIT compilation of a method and to pad them with `nop`s. This is the approach used in Jazz.

4.2.1.2 Reflexive Blocks Using transient instrumentation probes introduces a problem when a loop has a single basic block, as is shown in Figure 4.7. In a Java program, a *reflexive block* is a result of a `do...while` loop without any additional control flow transfers in the loop body. The issue arises because of the operations that occur when a probe is executed:

1. A fast breakpoint is hit in the basic block, jumping to the trampoline code.
2. The trampoline jumps to the payload.

3. The payload records coverage, if applicable.
4. The payload places probes for coverage successors and anchors.
5. The payload removes the fast breakpoint from the basic block.
6. The payload returns to trampoline.
7. The trampoline returns to the start of the basic block.

If one of the coverage successors or anchor probes is the block itself, the payload will replace this probe (unnecessarily, since it has not yet been removed) and then remove it. This situation is not a problem for node coverage since it does not affect the correctness of the coverage results. However, for branch coverage, there is now no way to record the coverage if the self-loop is taken.

There are several possible solutions for this issue. One approach is to split the basic block into two logical blocks, placing two probes, one to record the source and one to record the sink. If the block is not long enough for this, due to increasing the potential short block problem, another approach is to rewrite the branch instruction to jump to some instrumentation code, essentially instrumenting the back edge itself.

It is interesting to note that if steps 4 and 5 are reversed, the fast breakpoint will be in the basic block when control is returned to it, causing a jump back to the trampoline and payload in an infinite loop.

Pre-seeding eliminates this issue in the case of node coverage because no probe will place any other probes, including itself. However, branch coverage, even with pre-seeding will still have the reflexive block problem. The current implementation of DDST under Jazz does not handle reflexive blocks.

4.2.2 Trampolines

Each probe location (where a fast breakpoint is seeded or placed) has a corresponding trampoline location. A trampoline is an out-of-line function call, allowing for a smaller instrumentation probe, in this case, just a single relative jump. The trampoline contains code that is specific to each probe location and calls a payload function to do the work associated with the instrumentation probe.

Every trampoline starts with a context switch where the state of the machine registers is saved. For demand-driven node coverage, the trampoline then sets up the address of the entry in the PLT for recording coverage and placing successors (if not pre-seeding). Branch coverage has three types of trampolines: one for the regular basic blocks, another for singleton blocks, and a final one for stranded blocks.

Identically to the node coverage trampoline, the regular block payload passes the PLT entry for the breakpoint to the payload. For singletons, the PLT entry is passed so that the probe can place coverage and anchor successors and also explicitly passes the appropriate coverage counter to record the edge from the one and only control flow predecessor. The stranded block trampoline is similar to the singleton one, but instead of passing the edge counter, the counter representing that the stranded block is safe to remove is passed instead.

Every trampoline then invokes the appropriate payload function code. When that code returns, the trampoline restores the saved registers. It then determines the appropriate location in the original code to return to and executes the jump.

4.2.3 Payloads

Each trampoline has a call to a function payload that does the work of recording coverage, removing probes, and placing probes as necessary. As with the trampolines, there is a payload for demand-driven node coverage, and three more for demand-driven branch coverage.

4.2.3.1 Node Coverage Payload The payload for doing demand-driven node coverage is comprised of four actions:

1. Mark the coverage counter in the PLT as covered for this block (passed as a parameter from the trampoline),
2. Conditionally place probes in control flow successors,
3. Remove probe from the basic block currently being executed, and
4. Return to the trampoline.

The condition of step 2 is the coverage of the node where a potential probe would be placed. If pre-seeding is done, step 2 is omitted.

4.2.3.2 Branch Coverage Regular Payload The regular payload for branch coverage is similar to the one for node coverage. The same four steps are taken; however the first step, marking the covered edge, requires some additional work. A regular payload is assigned to a basic block that meets the previous *Fork* definition (otherwise it would be a singleton, by definition). The payload must determine which of the control-flow predecessors was executed immediately before this block. This is achieved through the previously hit probe variable that is stored in each thread's local storage. To mark the edge as covered, the PLT is examined to find the counter representing the edge between that predecessor and this probe. Thus, each probe in branch coverage needs an additional step, to be inserted as step 2:

2. Record this block in the previously hit block variable.

This step must be done prior to returning to the trampoline. Additionally, placing the control flow successor probes is predicated on the coverage of the edge between the current block and the successor being placed. Any anchor probes are predicated on a special per-stranded block value that is updated by the stranded block's payload when all incoming edges are covered.

4.2.3.3 Singleton Payload The singleton payload does not have to iterate through the PLT to find which counter is associated with the edge just executed. The counter's address is directly passed to the payload from the trampoline. Otherwise, the singleton payload is the same five steps as the regular payload of the previous section.

4.2.3.4 Stranded Payload The stranded block problem requires the control-flow predecessors of the stranded block to remain until all edges into the stranded block are covered. In the planning phase, a special predicate variable was allocated in the PLT to identify at runtime whether the anchor probes need to be replaced, without concern for the covered

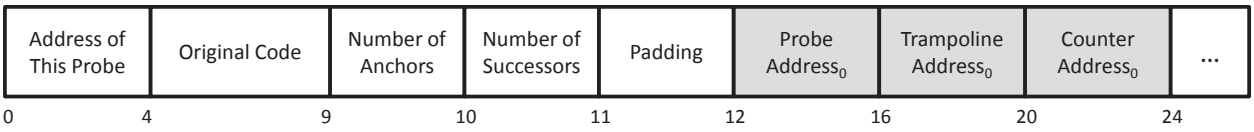

Figure 4.8: Row in the PLT for DDST.

or uncovered state. As part of the stranded block payload, this condition must be updated so that the probes involved in handling the stranded block problem can be eventually removed.

The stranded block trampoline passes the address of this special variable to the payload. Before the payload returns but after coverage has been recorded, the coverage of each incoming edge to the stranded block's is logically-ANDed together and the result stored in the variable.

4.2.4 Probe Location Table

Each payload depends on a data structure that encodes the test plan called the Probe Location Table or PLT. The PLT contains a table row for each probe that will be placed in the test region, even if that probe is not part of the seed set. The layout of a PLT entry is shown in Figure 4.8.

The first four bytes are the address of the current probe in the test region code. This location is stored so that the fast breakpoint can be removed. Removal is done by copying the original machine instructions over the fast breakpoint. The breakpoint is implemented as a 32-bit relative jump in x86, which is five bytes long. The table row is filled with the five bytes of original machine code prior to seeding.

Next are two values that indicate the amount of work the probe needs to perform at runtime. A probe will place the number of probes indicated, provided that they record an uncovered edge or are an anchor to an uncovered stranded block. The number of anchors is a separate entry because the stranded block solution requires anchors to remain until all incoming edges into a stranded block are covered. Thus, the anchors' coverage

counters, which predicate replacing the anchors, will all be set to "covered" at the same time—when the stranded block is covered. This work is performed in the stranded block payload.

Next is a single byte of padding that is unused. This makes the fixed size of the table row a multiple of four bytes for better cache alignment.

For each anchor and coverage successor there is a 12-byte entry, the first four bytes of which are the location place a fast breakpoint. Next is the offset to the trampoline for that particular probe. These four bytes are appended to the jump 32-bit relative opcode (`0xE9` in x86) to form the five-byte fast breakpoint. Finally, there is a pointer to the coverage counter that records the edge between the current probe and the probe to insert. The indirection is necessary for two reasons. First, for a regular stranded block, every anchor probes's replacement is predicated on the special stranded block counter, whose coverage is only set when all incoming edges into the stranded block are covered.

Additionally, a counter is also shared among more than one entry in the if-then stranded block solution. The counter that represents the coverage of edge $1 \rightarrow 3$ in Figure 4.2 is a predicate both to block 1 placing a coverage successor in block 3 as well as to block 3 replacing an anchor probe in block 1 (if only the path $1 \rightarrow 2 \rightarrow 3$ has been taken thus far).

4.3 EVALUATION

The current Jazz implementation is integrated with Jikes RVM 3.1.0. It was built for x86-64 Linux (BaseBaseMarkSweep RVM configuration). Both the RVM and applications to be tested are compiled (JIT'ed) without optimization. A mark and sweep garbage collector is used. All experiments were done on a lightly loaded quad-core Intel Xeon processor (2GHz with 4MB of L2 cache and 8GB RAM). The operating system is Red Hat Enterprise Linux Workstation release 4.

I used SPECjvm98 [11] for benchmark programs. The implementation does not include test techniques for multi-threaded programs, and as such, *mtrt* was excluded, as it is a

Table 4.1: Properties of the SPECjvm98 benchmark suite.

Benchmark	Methods	Nodes	Edges	Runtime (s)
check	79	662	789	1.76
compress	42	191	217	25.87
jess	436	1569	1594	24.19
db	27	220	299	30.18
javac	742	4584	5744	24.92
mpegaudio	201	793	802	18.75
jack	266	2019	2372	23.15

multi-threaded ray tracer. Details about the properties of the benchmarks are shown in Table 4.1. The methods column is the number of executed methods. Nodes and edges refer to the complexity of the control flow graph, with nodes being equivalent to the number of basic blocks. The runtime column is the baseline time of execution without structural testing. All timing results were determined by averaging three runs of each benchmark.

To evaluate Jazz and DDST, I present three sets of results for seven different coverage tests (Static Node, Static Node Agrawal, Demand Node, Demand Node Agrawal, Static Branch, Static Branch Agrawal and Demand Branch). First, the cost of structural testing is experimentally examined. This includes the cost of test planning and execution of the instrumentation. Next, the memory overhead for each instrumentation type used by the different tests is studied. Finally, how the tests interact and impact garbage collection is examined.

4.3.1 Performance Overhead

Because structural testing requires "extra work" when a program is executed, it imposes performance overhead. In the approach employed by Jazz, there are two sources of overhead. First, there is overhead due to test planning (i.e., identifying how and where to instrument the program) because the planning is done as part of JVM execution. As a result, the test planning overhead is fully observed by the user. Second, there is overhead

from the instrumentation to collect coverage information. This overhead is due to the complexity of the test and the implementation efficiency of Jazz.

Figure 4.9 gives the overhead for node testing (Static Node, Static Node Agrawal, Demand Node, and Demand Node Agrawal). Figure 4.10 gives overhead for branch testing (Static Branch, Static Branch Agrawal and Demand Branch). Performance overhead is reported as percentage increase in runtime over baseline performance without testing (see Table 4.1). The overhead reported in these figures comes from the number of probes needed for a particular test as well as the efficiency of Jazz's instrumentation code. The light blue part of each bar is the overhead from test planning and the dark blue part is the overhead from the instrumentation executed to gather coverage information.

The most apparent trend shown in the node coverage results of Figure 4.9 is that the second and fourth bars, corresponding to the addition of Agrawal's technique for static probe reduction, are typically the highest by far. This test technique does the most analysis of the seven since it has to construct the CFG and compute dominator information to minimize the number of instrumentation probes. The planning cost for the demand techniques is under 6% in all cases except *check*, which runs uninstrumented in about 1.8 seconds. As there are fixed costs in instantiating Jazz, this cost is not well-amortized in *check* due to the fact that there are many methods but practically no reuse of them.

Runtime instrumentation overhead is small, especially with the demand-driven approaches. Both static tests have higher than average overhead on *compress* and *mpegaudio*. These benchmarks are loop-intensive and leaving instrumentation—even small static probes—in the loop for the entire program execution is costly. The demand techniques do better on these programs; the overhead of instrumentation is only incurred once because the instrumentation is removed after a few loop iterations.

Branch testing, shown in Figure 4.10, has a similar trend in planning overhead as node coverage. The rightmost bar, representing Static Branch Agrawal, again shows the high cost of doing Agrawal's probe reduction technique. The Demand Branch and Static Branch (left and middle bars) show overhead under 25% except in *check* which runs too quickly to amortize startup costs.

Branch testing demonstrates the effect where the instrumentation overhead begins to

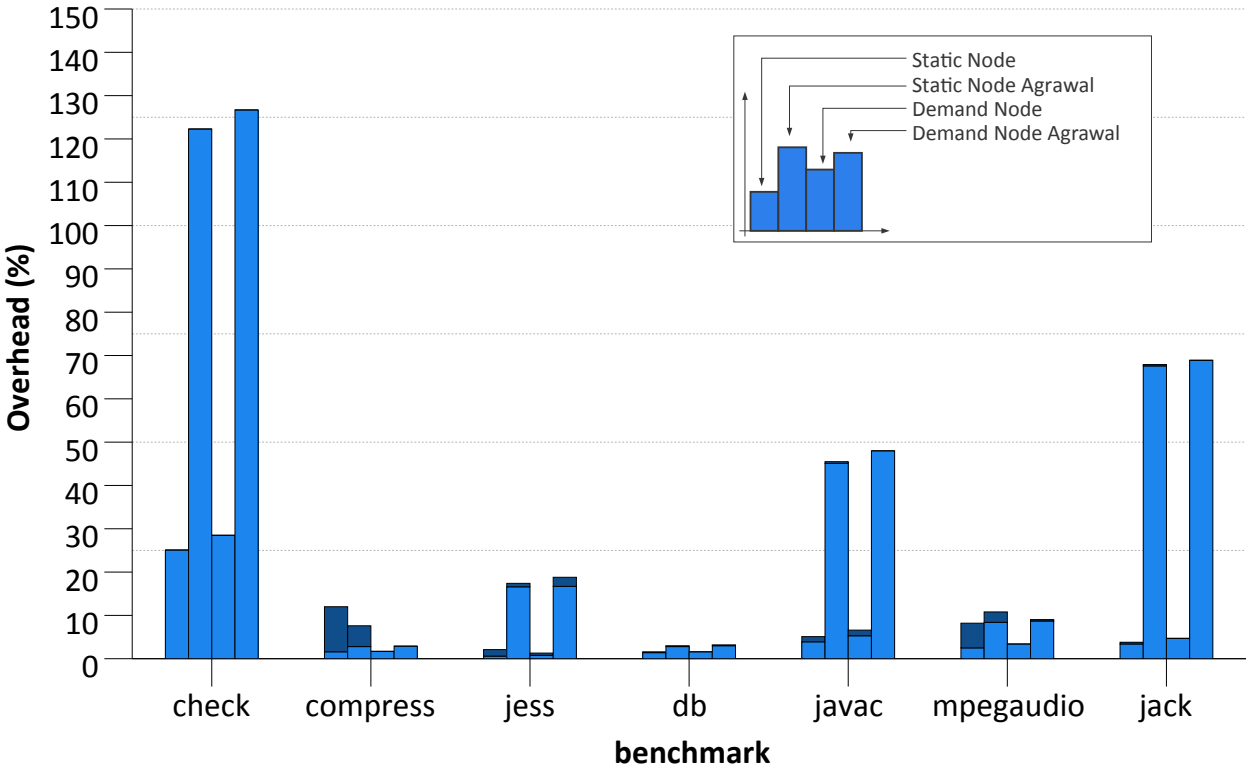

Figure 4.9: Node planning overhead (light blue) and instrumentation (dark blue).

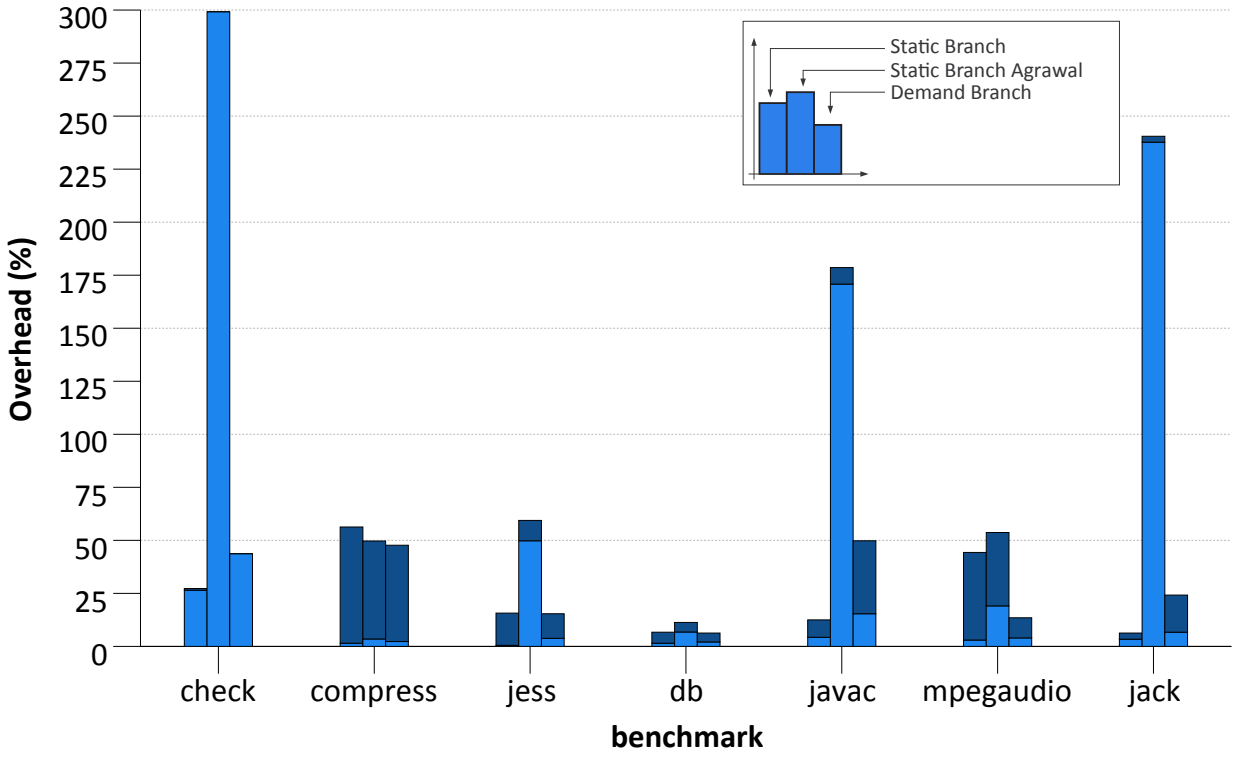

Figure 4.10: Branch planning overhead (light blue) and instrumentation (dark blue).

dominate total overhead. The loop intensive programs are the ones where demand-driven techniques are expected to do the best because instrumentation is quickly removed. In *mpegaudio*, this expectation is correct. However in *javac* and *jack* the demand-driven technique does worse than the static techniques. This result is due to two factors. The first one is that an individual instrumentation probe from the static technique is about 20 times cheaper than a demand-driven probe. This means that any region that is not re-executed frequently will incur unnecessary cost. The second issue is that the stranded block problem's solution requires demand probes to remain in the code until *all* incoming edges to the stranded block are covered (i.e., 100% coverage). If this condition happens late in program execution, or perhaps never, then it is less costly to use the inexpensive static probes for the full execution.

In the branch results no one technique is always the best. This result suggests that to get the minimal overhead for testing, a test engineer may want to determine a combination of techniques to apply together on different methods. With Jazz, this approach is easily supported—a test specification can be written to select the appropriate test for different methods.

These results show that the runtime cost of implementing test planners in Jazz is low for any program that runs long enough to amortize the small startup cost. For test planners that do a lot of work to determine where to place instrumentation, such as with Agrawal's algorithm, it may be beneficial to perform the planning offline or save planner results between runs when the code to be tested has not changed.

4.3.2 Memory overhead

Jazz supports memory allocation from the JVM's managed heap and the operating system (i.e., outside of the JVM). Table 4.2 lists the memory requirements for the tests in Jazz's test library. The figures in the table are the sum of the payload code, trampoline code, test plan, and inline instrumentation. The baseline GC heap size is 20 MB and can grow to a maximum of 100 MB.

The first four rows in the table detail the memory needs of the node coverage test

Table 4.2: Total memory usage for each of the SPECjvm98 benchmarks.

	check	**compress**	**jess**	**db**	**javac**	**mpegaudio**	**jack**
Static Node	9.8 KB	2.6 KB	19.7 KB	3.3 KB	66.0 KB	10.5 KB	29.7 KB
Static Node Agrawal	5.5 KB	1.4 KB	10.8 KB	2.0 KB	42.2 KB	5.5 KB	18.3 KB
Demand Node	19.7 KB	5.3 KB	39.3 KB	6.7 KB	132.0 KB	21.1 KB	59.4 KB
Demand Node Agrawal	11.0 KB	2.8 KB	21.5 KB	4.0 KB	84.4 KB	11.1 KB	36.6 KB
Static Branch	21.7 KB	6.2 KB	49.3 KB	7.5 KB	152.4 KB	24.9 KB	65.9 KB
Static Branch Agrawal	16.0 KB	4.4 KB	31.4 KB	5.9 KB	117.1 KB	16.6 KB	51.4 KB
Demand Branch	41.1 KB	10.2 KB	75.6 KB	14.5 KB	320.2 KB	37.6 KB	115.5 KB

implementations. As described earlier, the amount of work done by Static Node is small enough that a separate test payload is unnecessary. Instead, coverage information can be collected for Static Node with just four machine instructions (12 bytes) per node (basic block). Static tests do not use any trampolines. The test plan has only one word (4 bytes) to storage coverage result. The total memory usage ranges from only 2.6 to 66.0 KB. Static Node Agrawal, due to the probe reduction algorithm, needs even less space. The probes and test plans are the same size as in Static Node, but there is an average 43% reduction in probes, requiring only 1.4 to 42.2 KB of total storage.

Demand-driven node coverage (Demand Node and Demand Node Agrawal) have larger payloads, trampolines, and data storage requirements than the static tests. The demand-driven payload is seven machine instructions that are shared across all probes in all methods. Each fast breakpoint jumps to a six-instruction trampoline that transfers control to the payload. The demand test plan is larger than the static one because a location is reserved to store the instruction(s) overwritten when the fast breakpoint is inserted. This location holds the instruction so that it can be replaced when the probe is removed. Demand Node and Demand Node Agrawal require about twice as much memory as the static counterparts.

The final three rows show the memory requirements for branch testing. Since this coverage test must look up which edge should be recorded at runtime, it has a slightly larger payload than node coverage. Static Branch and Static Branch Agrawal share the same 40-byte payload and insert six machine instructions per node to jump to that

payload. Per incoming edge, the test plan needs a unique identifier for the potential CFG predecessor block and a location to record coverage. Static Branch requires 6.2 to 152.4 KB of total storage. Static Branch Agrawal reduces the number of probes by 11% on average and this translates into a 22% average reduction in memory size, with total usage ranging from 4.4 to 117.1 KB.

Demand Branch needs three payloads for the three specialized probe types (to solve the stranded block problem). The test plans are larger due to once again needing to store the original code that the fast breakpoint overwrote as well as an entry for each CFG successor (and CFG predecessor when dealing with a stranded block) to place at runtime. Each record has three words of storage: the address for placing the fast breakpoint, the offset of the trampoline to construct the relative jump from, and a counter that serves both to record the edge coverage and predicate if the probe should be placed. Despite the extra memory, the experiments show only a 2.1 times increase in memory usage over Static Branch.

The maximum memory usage for any benchmark was under one third of a megabyte. From these results, I conclude that the library of tests that Jazz provides all have reasonably small memory requirements.

4.3.3 Impact on Garbage Collection

Planning and performing a coverage test requires memory storage as shown in Table 4.2. The memory space is allocated from two places. The test plans and payloads are allocated outside of the Java heap with `mmap` (or `malloc`). The trampolines, inline instrumentation, and planner objects themselves are allocated from Java's heap. The RVM does not have a separate heap for "internal" Java code. As a result, Jazz's memory is intertwined with application memory, which can cause pressure on garbage collection. This adverse interaction can be significant and harm program performance. In my experiments, the average garbage collection took 459 ms (range: 295–2161 ms). This high cost clearly demonstrates that avoiding additional GCs is just as important as avoiding the execution of instrumentation probes.

Table 4.3: Change in the number of garbage collections due to testing with Jazz.

	check	**compress**	**jess**	**db**	**javac**	**mpegaudio**	**jack**
Forced GCs	4	2	2	2	6	2	2
Baseline Unforced GCs	0	7	9	4	4	0	9
Static Node	—	+1	-1	—	—	—	—
Static Node Agrawal	+1	—	+2	—	+3	+1	+4
Demand Node	—	—	-1	—	—	—	—
Demand Node Agrawal	+1	—	+1	—	+3	+1	+3
Static Branch	—	+1	—	—	—	—	—
Static Branch Agrawal	+3	+1	+3	+1	+10	+2	+8
Demand Branch	—	+1	—	—	+1	—	—

To analyze Jazz's influence on the number of garbage collections, I ran each of the seven test techniques with each benchmark to collect a verbose GC trace (provided by an argument to the RVM). This trace indicates when GC is invoked and the size of the collected heap. The results are shown in Table 4.3. The top two rows (in italics) give the number of collections for the baseline that does not do testing. These numbers report how many calls are made to `System.gc()` and how many GCs happen in an uninstrumented run (without testing). The other rows show the increase (+), decrease (-), or no change (—) in collections over the baseline for each test. The default initial heap size of 20 MB and maximum heap size of 100 MB were used. For the baseline, I measured the number of garbage collections in an uninstrumented run of each program. The default maximum heap size of 100 MB was used. The top two rows of Table 4.3 show the results separated by origin of garbage collection. As part of the SPECjvm98 benchmark harness, `System.gc()` is called immediately before and after each run of the core portion of each benchmark. This explains why there are two "forced" GCs per run. The additional forced GC calls for *check* is due to the benchmark being a test of JVM correctness and *javac* has a call between each file it compiles as part of its conversion to a benchmark.

The remaining table rows show the net change in unforced garbage collections for each test technique. Using Agrawal's technique causes many additional garbage collections due to the amount of temporary graphs constructed for analysis on each benchmark.

Once the seed set is placed during test planning, these temporary graphs are no longer needed. In general, the highest number of increased garbage collections happen on the benchmarks with the most methods, such as *javac* and *jack*. Since the test planner is invoked for each method, more memory space is needed, leading to more garbage.

An interesting effect occurs in *jess* with the static node and demand-driven node techniques. In both cases, the instrumented version has *one less GC* than the baseline. This situation is a result of the way that the RVM grows the heap when GC occurs. The amount the heap is increased depends on how recently the heap was grown. This changes the GC points and allows for fewer live objects to exist in later GCs, requiring the heap to grow less and fewer GCs to occur.

These results show that for the test types that do not depend on Agrawal's minimization technique, the impact on the number of GCs was acceptably small. When the work for Agrawal's support service is added in, there is a significant increase in memory usage that additionally impacts runtime from doing the additional GCs. Offline planning or caching planning results for unchanged methods may be necessary to get reasonable performance out of the technique.

4.4 SUMMARY & CONCLUSIONS

This chapter described an alternative to the statically-inserted instrumentation probes of traditional testing tools called Demand-driven Structural Testing. DDST seeks to remove instrumentation probes dynamically, allowing for the majority of the execution of a program to be instrumentation-free. DDST also supports dynamically inserting probes, avoiding placing instrumentation along paths that are infeasible or not exercised in a given test run.

The additional run-time work that a DDST instrumentation probe must perform requires a more sophisticated test planner than was necessary for Static or Static Agrawal. Node coverage testing under DDST involves placing a single seed probe in the entrypoint to the test region or to preseed every node in the region. These probes are implemented

as fast breakpoints and are removed as soon as they hit.

Branch coverage can also benefit from DDST, but since the instrumentation probes are placed in blocks rather than edges, certain control-flow structures such as the stranded block problem make both planning and the runtime actions performed by a probe more complex.

I implemented DDST for node and branch coverage under Jazz and evaluated their performance relative to Static and Static Agrawal. For node coverage, DDST is the clear winner. The instrumentation cost is amortized quickly as only the first execution of a basic block records coverage. Branch coverage's performance is affected by the higher cost of a DDST instrumentation probe compared to a static one and by the longer-lived probes necessary to solve the stranded block problem. Demand Branch performs best in loop-intensive code but can be worse than Static if code re-execution is infrequent or stranded blocks are common with coverage never reaching 100%.

5.0 PROFILE-DRIVEN HYBRID STRUCTURAL TESTING

DDST's RESULTS do not definitively answer the question of which test technique to use in practice. For node coverage, the results are clear: demand-driven removal of probes is the best choice, resulting in the lowest overhead. For branch coverage, however, the choice is not as clear. With certain programs or inputs, Static Branch or Static Agrawal Branch may result in lower overhead than DDST Branch.

When these three techniques are evaluated for branch coverage on the SPECjvm98 benchmark suite, the results show that there is no single best choice across all programs. Table 5.1 shows the overhead that each technique incurs over the baseline uninstrumented execution time when applied uniformly to all methods in the program. DDST incurs the lowest overhead most often, for four of the seven benchmarks. Static Branch's simple planner, low individual instrumentation probe cost, and its application to short-running benchmarks, results in it being best on the remaining three programs. Agrawal's algorithm is so expensive that it is never the best due to the cost of test planning never

Table 5.1: No single test technique is consistently the best.

	Static	Agrawal	DDST
check	**27.8%**	301.1%	44.3%
compress	56.3%	49.8%	**47.9%**
jess	16.5%	59.7%	**15.9%**
db	7.0%	11.2%	**6.3%**
javac	**12.8%**	180.7%	50.4%
mpegaudio	44.1%	54.4%	**13.3%**
jack	**7.0%**	242.3%	24.3%

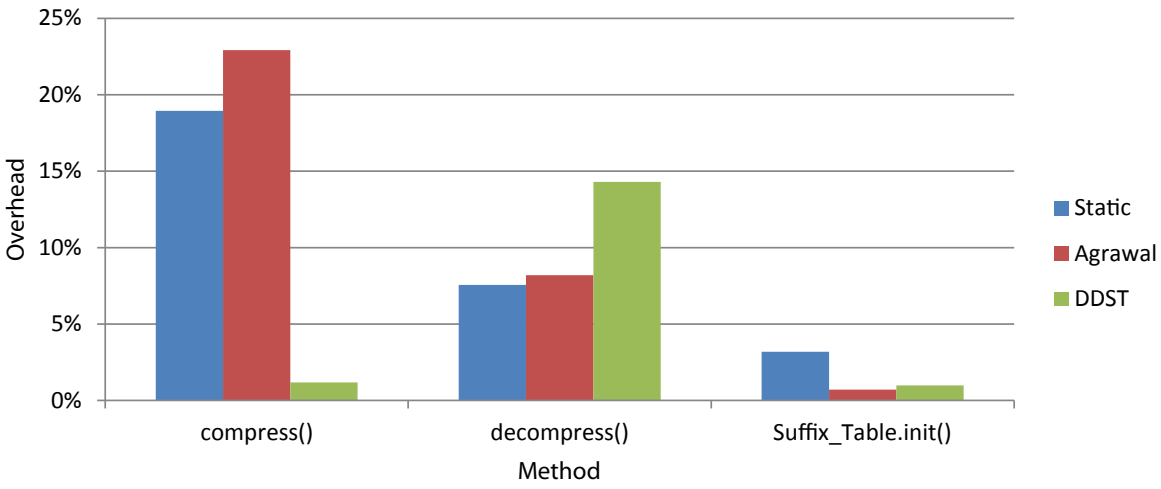

Figure 5.1: Three methods from *compress*.

being amortized by the reduction in probes.

DDST's strength is that its instrumentation probes can be removed quickly and the majority of the program will execute without the overhead of the instrumentation. DDST should then be expected to perform the best on loop-intensive benchmarks, such as *compress* and *mpegaudio*, which matches the results shown in Table 5.1.

However, just as every program does not behave in the same way, neither does every method in a program. Finding the best choice on a per-method granularity shows the same trend as before: *No one technique is uniformly the best*.

Figure 5.1 illustrates this point with an example from *compress*. This benchmark compresses five files five times and then decompresses them to get the original files. Both the `compress` and `decompress` functions are similar in structure with a high-iteration loop reading the file data, and either adding or checking each codeword in a hashtable. However, as the figure shows, when instrumentation is added to the two methods, the best test technique is dramatically different. The expectation of DDST performing best for loop-intensive code holds true in `compress` but fails in `decompress`. A closer examination of the source code shows that the decompress loop has an error check for an invalid codeword that is never encountered in the benchmark's run—resulting in an

uncovered stranded block.

Although Agrawal's technique was never the overall best when applied to a whole program, the results for the `Suffix_table.init` method show it can be the best on a per-method basis.

5.1 SOURCES OF DDST COST

The performance of DDST Branch is complicated by two factors. First, individual DDST instrumentation probes are more expensive to place and remove than static, compiler-inserted probes. This is why DDST performs well on loop-intensive programs: It uses the large number of instrumentation-free iterations to amortize the cost of a probe. For methods or programs that do not have enough instrumentation-free executions, the permanent static instrumentation ends up incurring less cost overall.

The second issue is that even in loop-intensive code such as the `decompress` method, the solution to the stranded block problem requires some probes to remain despite the coverage information recorded by these probes having already been determined. This section will explore the impact of these factors in contributing to the cause of DDST's inconsistent performance.

5.1.1 Instrumentation Probes

The payload for DDST Branch is made up of more instructions than the Static Branch payload because a DDST instrumentation probe has to place additional probes during execution. Table 5.2 shows the x86 instructions that comprise the Static Branch payload. There are 14 instructions that take up 40 bytes of space. Comparing it to the DDST payload shown in Table 5.3, the DDST payload requires almost three times as many instructions with a similar increase in space.

The major difference in the two payloads is the second loop in DDST that places the coverage and anchor probes, labeled as `placeSuccessorFastBreakpoints`. Both

Table 5.2: Size and code for a Static Branch probe.

	Label	Machine Code	Offset
1	initializeFramePointer:	mov ebp, dword ptr [esi + 140]	0
2	getPreviousHitBrkpt:	mov eax, dword ptr [ebp - 8]	6
3	setPreviousHitBrkpt:	mov dword ptr [ebp - 8], edx	9
4	markCounterAsHit:	mov ecx, dword ptr [edx]	12
5		test ecx, ecx	14
6		jz return	16
7		add edx, 4	18
8	loop1:	cmp ebx, dword ptr [edx]	21
9		jne next	23
10		mov dword ptr [edx + 4], 1	25
11		jmp return	32
12	next:	add edx, 8	34
13		loop loop1	37
14	return:	ret	39

this loop and the one that records coverage (`markCounterAsHit`) will execute for each CFG successor, which was previously reported as 1.3 basic blocks with a variance of 2.0 blocks. With three times the number of instructions and twice as many loops, a reasonable estimate would be that a DDST probe might have a dynamic instruction count around six times that of Static and perhaps a similar increase in cost.

To test this, the probe costs for Static and DDST were collected on SPECjvm98. Table 5.4 shows C_{static} and C_{demand}, the costs of a single Static probe and a DDST probe. The times were computed from two pieces of information: the number of instrumentation probes executed at runtime and the time spent executing them. To obtain the number of probes, a modified version of Jazz inserted special counting instrumentation probes that recorded the number of times each was hit and reported the values at the end of the run. The cost of instrumentation was determined by finding the difference between two runs: the first where the normal instrumentation probes were inserted and the second where planning was done but no instrumentation was inserted.

The first two columns of Table 5.4 are the cost per probe calculated as described. The ratio of how much more expensive a demand-driven probe is than a static probe is in the next column, labeled ρ. On average across the benchmarks, a single DDST

Table 5.3: Size and code for a regular DDST Branch probe.

	Label	Machine Code	Offset
1	initializeFramePointer:	mov ebp, dword ptr [esi + 140]	0
2	getPreviousHitBrkpt:	mov eax, dword ptr [ebp - 8]	6
3	setPreviousHitBrkpt:	mov dword ptr [ebp - 8], edx	9
4	markCounterAsHit:	test eax, eax	12
5		jz removeFastBreakpoint	14
6		movzx ecx, byte ptr [eax + 9]	16
7		add eax, 12	20
8		test ecx, ecx	23
9		jz removeFastBreakpoint	25
10	loop1:	mov ebx, dword ptr [edx]	27
11		cmp dword ptr [eax], ebx	29
12		jne next	31
13		mov ebx, dword ptr [eax + 8]	33
14		mov dword ptr [ebx], 1	36
15		jmp removeFastBreakpoint	42
16	next:	add eax, 12	44
17		loop loop1	47
18	removeFastBreakpoint:	mov eax, dword ptr [edx]	49
19		mov ecx, dword ptr [edx + 4]	51
20		mov dword ptr [eax], ecx	54
21		mov cl, byte ptr [edx + 8]	56
22		mov byte ptr [eax + 4], cl	59
23	placeSuccessorFastBreakpoints:	movzx ecx, byte ptr [edx + 10]	62
24		add edx, 12	66
25		cmp ecx, 0	69
26		je return	72
27	loop2:	mov eax, dword ptr [edx + 8]	74
28		cmp dword ptr [eax], 1	77
29		jge dontplace	80
30		mov eax, dword ptr [edx]	82
31		cmp byte ptr [eax], 0E9h	84
32		je existing_jmp	87
33		mov byte ptr [eax], 0E9h	89
34	existing_jmp:	mov ebx, dword ptr [edx + 4]	92
35		cmp dword ptr [eax + 1], ebx	95
36		je dontplace	98
37		mov dword ptr [eax + 1], ebx	100
38	dontplace:	add edx, 12	103
39		loop loop2	106
40	return:	ret	108

Table 5.4: Cost of an instrumentation probe for Static and DDST.

	C_{static}	C_{demand}	ρ
check	-7μs	-331ns	0.045
compress	12ns	223ns	17.5
jess	13ns	290ns	21.7
db	6ns	204ns	29.5
javac	23ns	350ns	15
mpegaudio	10ns	219ns	20.3
jack	15ns	345ns	23
Average			18.2

instrumentation probe for branch coverage is 18.2 times as expensive as an equivalent static probe.

With *check*, the addition of instrumentation probes results in faster execution than the planning-only baseline used to determine the probe cost due to GC effects. This yields a speedup that is shown in Table 5.9 as a negative instrumentation probe cost. Since *check* runs for such a short amount of time, GC overshadows the cost of instrumentation.

A DDST probe being 18.2 times as expensive as a Static probe is a greater difference than can be explained just by the dynamic instruction count. Instead, consideration must be given to the actions that a DDST probe performs: Namely, placing probes in the instruction stream at runtime.

5.1.2 Architectural Impact

Inserting or removing a DDST probes during runtime falls under the category of self-modifying code. Modern CPU architectures heavily cache both code and data to the point where self-modifying code can result in a variety of performance penalties.

Figure 5.2 shows the results of using Hardware Performance Counters [34] to collect the instruction cache (i-cache) miss rate on SPECjvm98 for DDST Branch and Static Branch. The results are shown as a change relative to a baseline uninstrumented run.

DDST shows a consistent increase in i-cache misses, including an increase on *compress* of over 66 times the baseline miss rate. Remarkably, *compress* is one of the programs

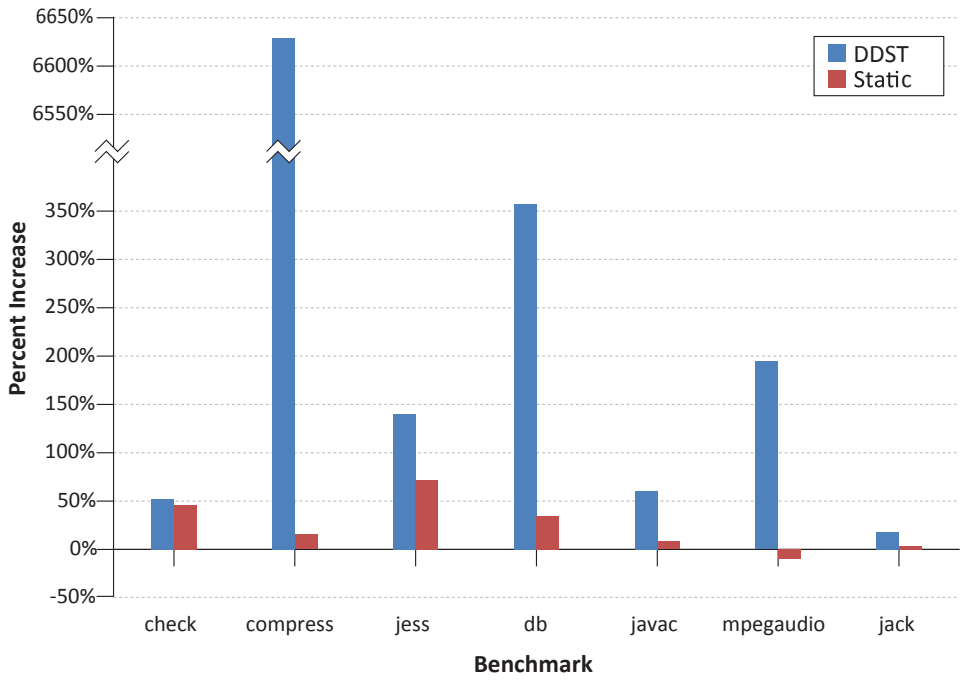

Figure 5.2: Change in instruction cache miss rate for DDST and Static.

where DDST manages to perform better than Static, along with *jess*, *db*, and *mpegaudio*. These four programs have the highest increase in i-cache misses. The misses are a result of cache invalidations flushing the i-cache when a probe is inserted or removed.

Static also has an increase in i-cache misses over the baseline except in the case of *mpegaudio*. This is probably due to the increased pressure in the i-cache from inserting the trampolines used by Static inline into the instruction stream, resulting in growing the code by four extra instructions per basic block.

For the three programs where Static outperforms DDST (*check*, *javac*, and *jack*), both Static and DDST show similar increases in i-cache misses. This can be explained by the actions of the DDST probe. The DDST probe was designed to avoid placing an instrumentation probe when one is already at the target, eliminating unnecessary cache invalidations. Instructions 31 and 35 from Table 5.3 perform this check. Thus, it is likely that on these three programs that DDST probes are not being inserted or removed as often as in the other programs. This would result from either a lack of code re-execution

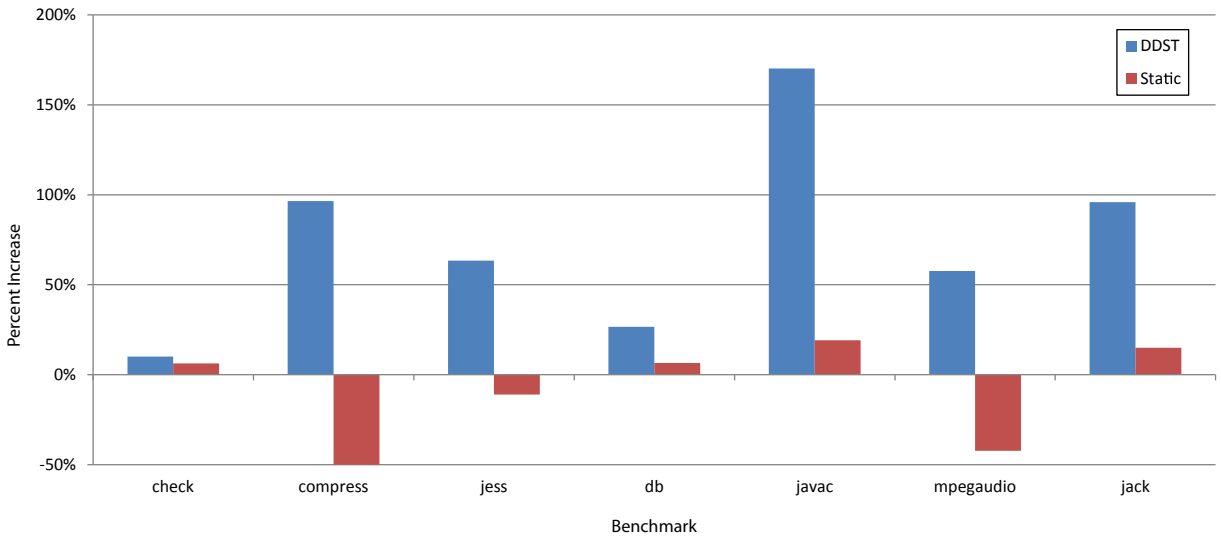

Figure 5.3: Change in branch misprediction rate for DDST and Static.

or from uncovered stranded blocks.

Other microarchitectural effects may result from the nature of DDST's instrumentation. Figure 5.3 shows the change in the branch misprediction rate on SPECjvm98. These results show modest increases, and, once again, DDST is worse than Static. This is likely due to Static's payload having one less loop and fewer branch instructions (two versus seven in DDST). The branches in the DDST payload are dependent on the number of predecessors, the number of probes to place, the coverage recorded so far, and the presence of a probe at a location already—factors that can vary greatly from probe to probe and even for a single probe during program execution.

In addition to the code effects of instrumentation, both Static and DDST increase the amount of data in a process's working set with the test plans necessary to drive and record coverage information. Figure 5.4 shows the change in the data TLB miss rate for both Static Branch and DDST Branch. For x86, DTLB misses include speculative accesses [34], and the interruption of a data processing loop, such as in *compress* and *mpegaudio*, to perform instrumentation may be harming locality and the effectiveness of prefetching.

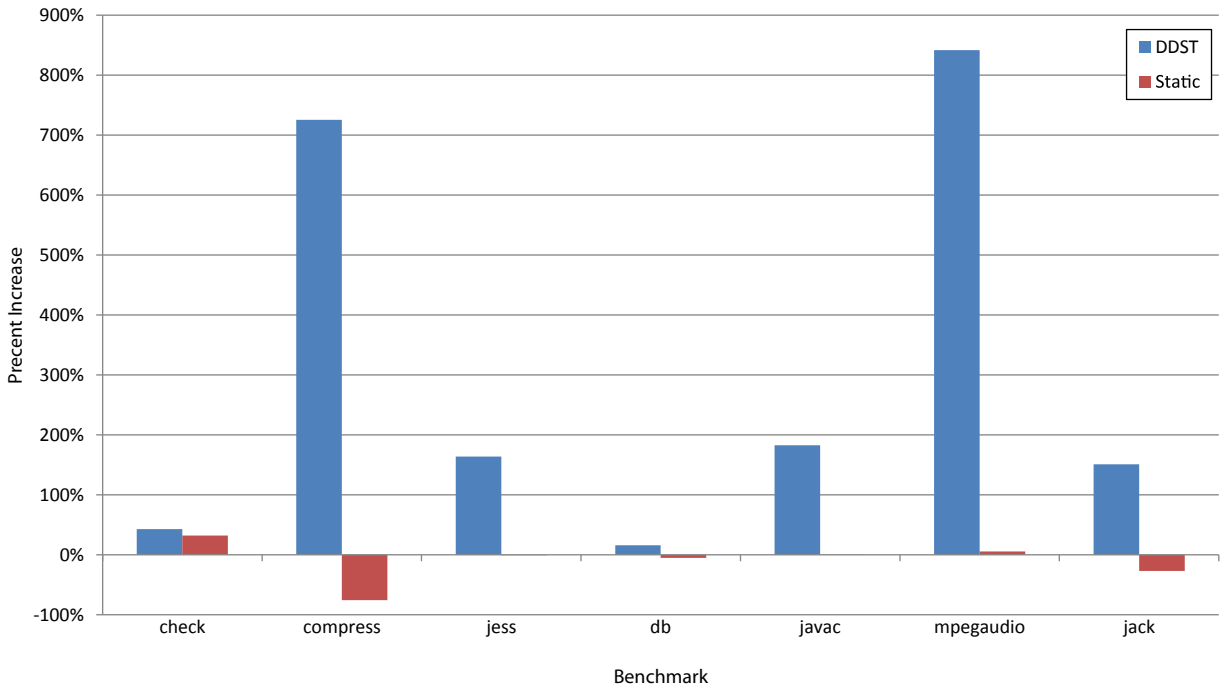

Figure 5.4: Change in DTLB miss rate for DDST and Static.

Table 5.5: Stranded blocks in SPECjvm98.

			Regular Stranded	
	Total Stranded	**If-Then Stranded**	**Count**	**Coverage**
check	90	34	56	12.5%
compress	34	16	18	50.0%
jess	198	57	141	27.7%
db	48	21	27	29.6%
javac	743	163	580	32.6%
mpegaudio	77	17	60	46.7%
jack	261	159	102	42.2%

There will also be DTLB effects from using self-modifying code. The write to a code page will require fetching that page into the data cache, and a new entry in the DTLB will be created, increasing pressure and potentially evicting other useful DTLB entries.

5.1.3 Stranded Blocks

While the previous sections have shown that DDST's instrumentation probe, the insertion and removal of probes, and the memory for the test plans all contribute to the cost of DDST, the overhead is amplified by the stranded block problem. If a DDST probe can be removed early in the execution of a method, its overhead will be amortized by the instrumentation-free executions when Static would still be incurring cost.

When this tipping point between DDST and Static is not reached, it must be because of one of two reasons:

1. The region under test is infrequently executed, or
2. Anchor probes remained due to an uncovered stranded block.

If the reason is the former, DDST will never be the best choice and Static should be applied if this situation can be anticipated.

To determine the impact of the stranded block problem, statistics on how frequently it occurs in SPECjvm98 are shown in Table 5.5. Take, for example, *compress*. There are a total of 34 stranded blocks, of which 16 are handled with the special If-Then stranded block

solution that mitigates some of the anchor probes necessary in the general stranded block solution. Of the remaining 18 "regular" stranded blocks, only nine (50%) are covered at runtime. This situation is the cause of the results shown in Figure 5.1, when the best test technique was different for the similarly-structured `compress` and `decompress` methods.

However, *compress* has the highest regular stranded block coverage at 50%, followed closely by *mpegaudio* at 42.2%, and DDST performs best on these programs. The third highest coverage is on *jack*, but Static has lower overhead than DDST, likely due to it being a parser generator with many methods but little iteration or code reuse.

5.2 SELECTING A TEST TECHNIQUE

DDST probes are unlikely to get significantly cheaper or shorter-lived due to the stranded block problem and, for some methods or programs, Static or Static Agrawal will be the best test technique to use. This presents an opportunity to target the strong points of each test technique and combine multiple techniques in a single program run. This should result in a mix that is better than the uniform application of any one test. For example, a method that has a stranded block that is never covered in a test run should be tested with Static as it is unlikely to benefit from DDST.

Since the coverage of a test region depends on the input data, it is impossible in the general case to perfectly predict *a priori* if a stranded block will be covered quickly or even at all. The only way to find out for certain is to run the program on the input and find out.

5.2.1 Average-driven Search

A direct approach to determine how to choose the best technique for each method in a program is to run them all and find the best one.

Every method in the benchmark suite was run with the three test techniques individ-

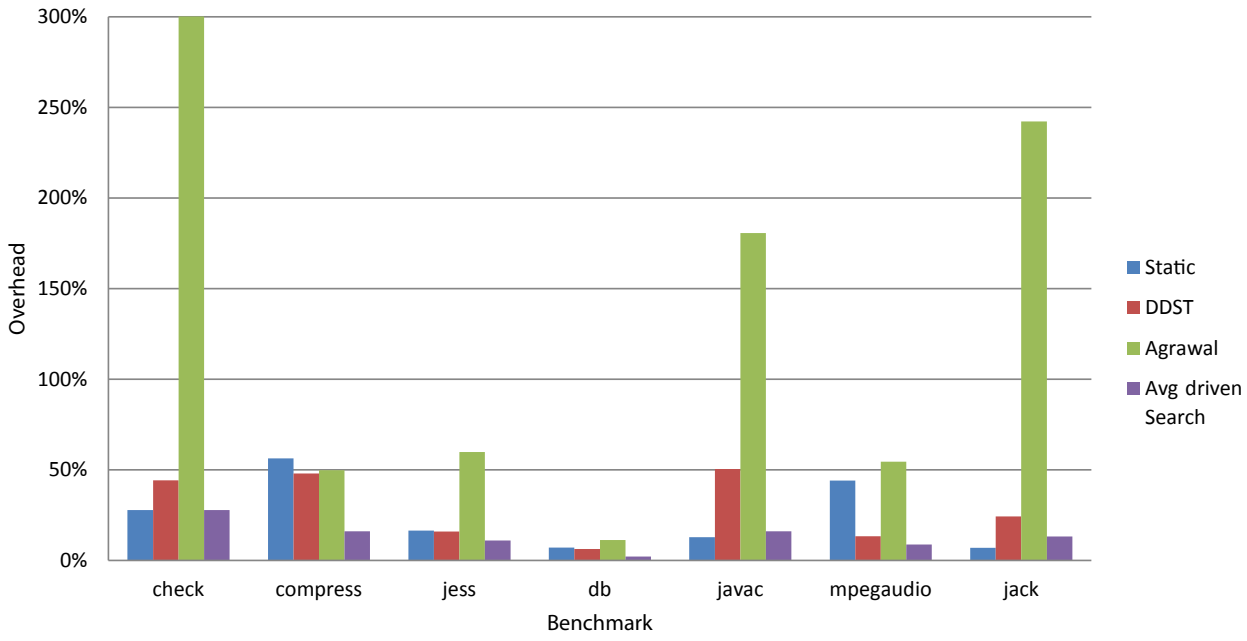

Figure 5.5: Average-driven search.

ually and the technique that performed best on the average of three runs was selected for each method. The resulting best choices from this "average-driven search" were then combined into a single test run. Figure 5.5 shows the observed overhead versus the uninstrumented baseline. Also shown are the results from the uniform application of a single test technique.

Average-driven search successfully finds a mix of test techniques that is the lowest in overhead for *compress*, *jess*, *db*, and *mpegaudio*. The biggest improvement occurs for *compress*, where the average-driven search discovers which stranded blocks are covered early and which are not.

Surprisingly, Figure 5.5 shows several occasions where the search was not successful in finding a better mix than the uniform application of a single method. Both *javac* and *jack* show this phenomenon. This result can be explained partly as an interaction with garbage collection (GC), as the precise time that GC is triggered affects how large the heap grows and how frequently subsequent GCs occur. When instrumenting a single

Table 5.6: Time required to perform average-driven search.

	Time (hours)
check	0.3
compress	2.7
jess	25.1
db	2.0
javac	46.1
mpegaudio	9.4
jack	15.5
Total:	*101.1*

method, the temporary objects created are unlikely to cause enough pressure on the heap to perturb GC the way a full testing run would. The imprecision of timing, rounding error in the calculations, and microarchitectural interactions may also affect the results, as these are two of the biggest benchmarks in terms of the number of methods.

While the workload of doing this search is embarrassingly parallel, it still involved running 1783 methods nine times each (three runs of the three test techniques). Table 5.6 shows the cost of running the search in terms of computer-hours of work. While *check* only required an 18 minute search, *javac* needed 46.1 computer-hours to run. The search over the entire benchmark suite took 101.1 computer-hours in total.

It is clear from the time requirements that finding a good overall test configuration this way is infeasible for reasonably-sized programs that are undergoing the typical testing and development cycle. To make matters worse, after doing this search, the resulting configuration is not even optimal. The results do, however, show that a hybrid approach to branch coverage testing can result in better overall performance than a naïve application of one technique to an entire program.

5.2.2 Profile-driven Test Selection

Although average-driven search is slow and inaccurate, it still validated that a mix of test techniques can outperform any single one. Discovering a good mix by an alternative

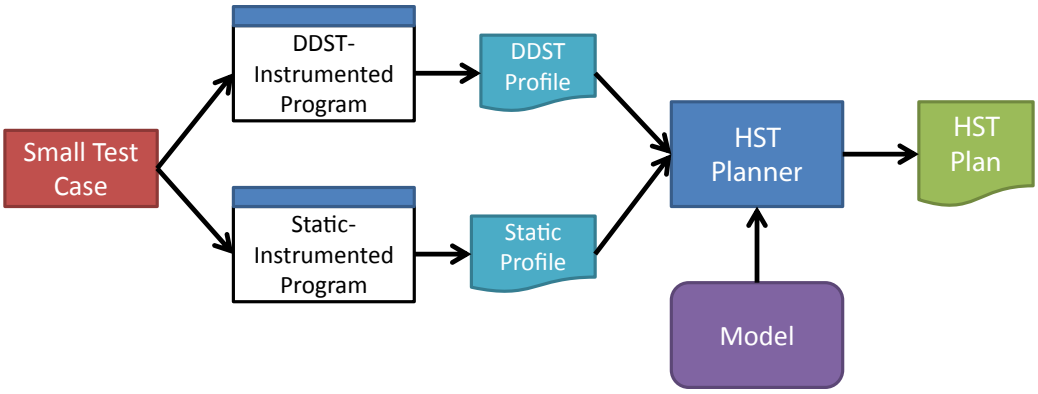

Figure 5.6: Hybrid structural testing via profiling.

approach that is faster may actually be practical to use as part of the testing process.

The two key factors that prevent DDST from being the best are that a single DDST instrumentation probe is more expensive than a static probe and that, in some instances, stranded blocks or infrequent execution of code do not allow the additional probe cost to be amortized away. With the costs of a single DDST and static instrumentation probe, a decision of which test technique to use could be computed from the reduction in dynamic probe executions resulting from DDST.

This is also a property that cannot be known *a priori*, but instead of running the whole program on the full input as done with the average-driven search, the runtime characteristics of a program can be discovered by profiling. Profiling runs the program with a small test case and collects dynamic probe counts for the various instrumentation techniques. The ideal is to do a small number of profile runs and then use the results to guide future test technique decisions made on the same program, possibly even after its code has been slightly changed.

5.3 DESIGN OF A PROFILE-BASED TEST SELECTOR

Profile-driven Hybrid Structural Testing (HST) is based upon knowing three aspects of the runtime properties of a region to test:

1. the number of dynamic probe executions saved using DDST or Agrawal,
2. the cost of planning, and
3. the cost of a single probe of each instrumentation type.

Figure 5.6 shows how to use profiling to determine the dynamic probe executions saved with DDST. A small test case is generated and the program is instrumented using Jazz with DDST. Additionally, a traditional profile of block frequency is collected, as this corresponds to the number of static probes executed. While the figure shows this as two separate runs, it is possible to combine them into a single execution of the program.

Static Agrawal Branch does not need its own execution run. The instrumentation probes are the same as Static Branch; however, Agrawal's algorithm simply places them in fewer locations. With the full block profile from Static, the number of probe executions that Static Agrawal will incur can be determined offline without a dedicated profile run.

The results of the two dynamic probe count profiles are fed into an HST Planner. The HST Planner incorporates a model that describes the expected runtime costs associated with testing. Using the profiles and the model, the planner then produces an HST Plan in *testspec*, the test specification language that Jazz takes as input. The plan specifies which test technique to use on each method as best determined by the HST Planner.

Figure 5.7 shows the overall framework for HST. Central to the testing process is the application to be tested, run under a JVM supporting Jazz. Additionally, there is a *test suite* of inputs that have been selected to exercise the application code. The presence of a test suite provides an opportunity to amortize the cost of profiling and generating the HST Plan since the same plan can be used for each test case in the test suite. Jazz's *test driver* interprets the *testspec* language of the HST Plan and applies the specified tests to the application code as it runs.

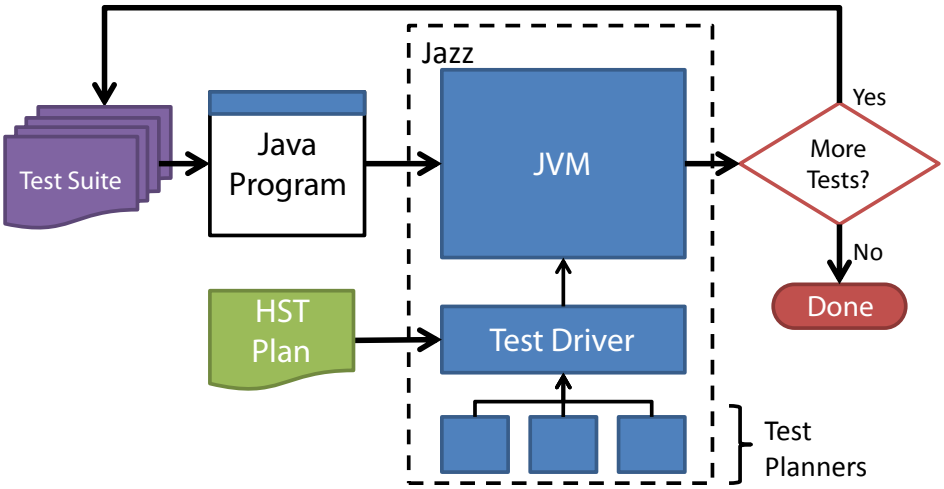

Figure 5.7: The overall framework for doing Hybrid Structural Testing.

5.3.1 Modeling Branch Coverage Testing

For profiling to choose the appropriate test technique, branch coverage testing's costs must be modeled accurately enough to choose among the three techniques implemented in Jazz. The overhead of testing comes from two main sources. The first is from the cost of executing an instrumentation probe to record the coverage information. The second is the cost of determining where to place probes, i.e., planning. This allows for the construction of a simple mathematical model to describe the cost of testing, C_{test}:

$$C_{test} = C_{probe} \times N_{probes} + C_{plan}$$

where, for a given technique, C_{probe} is the cost of a probe, N_{probes} is the number of probes hit during execution, and C_{plan} is the cost of planning.

Table 5.7 shows the models applied to three branch coverage techniques implemented in Jazz: Static, Static Agrawal, and DDST. The first column represents the values of C_{probe}, the cost of a single probe. This value is dependent on several parameters. The runtime cost of an instrumentation probe depends on the number of control-flow successors and/or predecessors. This value is fairly small and low in variance in most structured

Table 5.7: Models for Jazz's three branch coverage techniques.

	$\mathbf{C_{probe}}$	$\mathbf{N_{probes}}$	$\mathbf{C_{plan}}$
Static	$C_{static\ probe}$	Profiled	0
Agrawal	$C_{static\ probe}$	$\alpha \times N_{static}$	$C_{Agrawal\ Plan}$
DDST	$C_{demand\ probe}$	Profiled	0

programs. For SPECjvm98, the average number of successors per basic block is 1.3 with a standard deviation of 2.0. This allows for a single value to be used for all programs. Another aspect of probe cost is the implementation of a probe and the architectural effects of that choice. Demand-driven probes are based upon self-modifying code and static probes are inlined into the executable code, possibly increasing instruction cache pressure. With these issues in mind, the probe cost can be experimentally determined.

The dynamic probe count for Static and DDST must also be experimentally determined. However, there are two possibilities for finding the dynamic probe count under Static Agrawal Branch. The first possibility is to run the Agrawal test planner and determine where probes should be placed. A run of the resulting code is not necessary since Agrawal simply uses a subset of Static's probes. The results from Static can be filtered to include only those instrumentation probes that Agrawal would keep.

While this is feasible and will give an exact answer for Agrawal, it increases the amount of work that must be done in the HST Planner, which makes it more difficult to amortize the cost. As shown in Table 5.1, the increase in runtime for Static Agrawal is attributable to the cost of test planning since the Static instrumentation probes are used, and will have the same or fewer dynamic executions.

Instead of doing the work of Agrawal's algorithm in the HST Planner, the expected reduction in dynamic probes is computed by scaling the dynamic probe count in Static Branch by a factor α. This constant can also be experimentally determined.

While the HST Planner avoids the work of planning for Agrawal, test planning is still a source of runtime delay, and thus it must be a component of the model. As shown in Figures 4.9 and 4.10, the cost of planning for Static or DDST is not a significant source of

```
1: if N_static > ρ × N_demand then
2:   do_demand(m);
3: else if α × N_static × C_static > AgrawalPlannerCost(m) then
4:   do_agrawal(m);
5: else
6:   do_static(m);
7: end if
```

Algorithm 5.1: HST Planner

overhead and can be ignored. However, the overhead of Agrawal's algorithm is significant enough that it must be considered to accurately choose when the technique will be the best to apply. Agrawal's algorithm is quadratic in the number of edges in the CFG of the test region, yielding the model:

$$C_{Agrawal\ Plan} = k \times Edges^2 + c$$

where k is a constant representing the amount of work done for each edge and c is an additional constant overhead.

5.3.2 The HST Planner

The HST Planner determines how best to instrument each method encountered in the profiling run. It is shown in Algorithm 5.1. Line 1 determines when it is appropriate to use DDST from the relative cost of a single demand probe to a static probe, $\rho = \frac{C_{demand}}{C_{static}}$. If the savings are at least $\frac{1}{\rho} \times N_{static}$ static probes, then the planner predicts that the method m will perform best with DDST.

If DDST will not save enough dynamic probe executions to recoup the cost of the more expensive probes (compared to static), the planner chooses between static and static with Agrawal's reduction algorithm. Line 3 uses the model of the cost of the Agrawal planner and compares that to the anticipated reduction of dynamic probes via the factor α.

If the cost of the additional planning can be recouped by the reduction of probes, then the planner chooses Agrawal's technique for m.

If both DDST and Agrawal are too costly, for example in a method that is invoked only once or has a very complex CFG, the HST Planner chooses the Static technique.

5.3.3 Test Specification

The HST Planner constructs a test specification for each method encountered in the execution of the profile run. However, because the profile run's input is small, there may be methods that a full run encounters that were not part of the profile. For these methods, the plan contains a default rule that instruments unprofiled methods via the static technique. Static is chosen for two reasons. First, it is the best method in the majority of cases, and second, its planning cost is low.

Figure 5.8 shows an example test specification for *compress*. Each method encountered in the profile of the program is listed with the test technique to perform. The final line uses a wildcard to instrument any unseen methods using Static.

5.4 INSTANTIATING THE MODEL FOR JAZZ

The model has several parameters (shown in Table 5.8) that must be experimentally determined to decide for a method what test technique to apply. The relative cost of a DDST instrumentation probe versus a static probe is necessary to determine if the reduction in dynamic probe count is enough to justify using the more expensive DDST probe. Agrawal's algorithm also allows for a reduction in dynamic probe count by allowing for some coverage to be inferred rather than directly recorded. However, the cost of Agrawal's planner is high. To decide if Agrawal is the right choice, an estimate of the reduction is necessary and an accurate model of planner cost must be made.

```
spec.benchmarks._201_compress.Decompressor$De_Stack:is_empty:DEMAND_BRANCH
spec.benchmarks._201_compress.Input_Buffer:readbytes:DEMAND_BRANCH
spec.benchmarks._201_compress.Compressor$Hash_Table:clear:DEMAND_BRANCH
spec.benchmarks._201_compress.Compressor:cl_block:DEMAND_BRANCH
spec.benchmarks._201_compress.Harness:inst_main:STATIC_BRANCH
spec.benchmarks._201_compress.Input_Buffer:getbyte:DEMAND_BRANCH
spec.benchmarks._201_compress.Harness:fill_text_buffer:STATIC_BRANCH
spec.benchmarks._201_compress.Decompressor:<init>:STATIC_BRANCH
spec.benchmarks._201_compress.Compressor:<init>:STATIC_BRANCH
spec.benchmarks._201_compress.Compressor:output:DEMAND_BRANCH
spec.benchmarks._201_compress.Decompressor:decompress:STATIC_BRANCH
spec.benchmarks._201_compress.Decompressor:getcode:STATIC_BRANCH
spec.benchmarks._201_compress.Decompressor$Suffix_Table:init:DEMAND_BRANCH
spec.benchmarks._201_compress.Output_Buffer:writebytes:DEMAND_BRANCH
spec.benchmarks._201_compress.Code_Table:clear:DEMAND_BRANCH
spec.benchmarks._201_compress.Main:runBenchmark:STATIC_BRANCH
spec.benchmarks._201_compress.Compress:spec_select_action:STATIC_BRANCH
spec.benchmarks._201_compress.Harness:run_compress:STATIC_BRANCH
spec.benchmarks._201_compress.Compressor:compress:DEMAND_BRANCH
spec.benchmarks.**:=all:STATIC_BRANCH
```

Figure 5.8: Test specification for *compress* from a size 1 profile run.

Table 5.8: Parameters for the HST models.

Parameter	**Description**	**Origin in the Model**
α	Reduction in dynamic probe count due to Agrawal	Experimentally Determined or Profiled
ρ	Relative cost between Static and DDST probe	Experimentally Determined
C_{static}	Cost of a static probe	Experimentally Determined
C_{demand}	Cost of a DDST probe	Experimentally Determined
$C_{Agrawal\ Planning}$	Predicted cost of Agrawal Planning	Modeled or Experimentally Determined
N_{static}	Number of Static probes executed at runtime	Profiled
N_{demand}	Number of DDST probes executed at runtime	Profiled

Table 5.9: Costs incorporated into the HST model.

	Static Probes	Demand Probes	Agrawal Probes	α
check	67,310	60,290	56,887	15.5%
compress	1,034,903,676	48,482,793	994,151,120	3.9%
jess	253,190,572	11,435,239	187,395,337	26.0%
db	130,245,027	5,790,123	104,241,726	20.0%
javac	145,958,252	21,274,800	132,249,797	9.4%
mpegaudio	687,948,449	7,903,815	609,012,496	11.5%
jack	31,310,790	10,178,424	30,608,518	2.2%
Average				12.60%

5.4.1 Instrumentation Probe Cost

The cost of instrumentation probes was explored in Section 5.1.1. Table 5.4 showed the average time necessary for a single probe in Static and DDST, as well as ρ, the relative cost of a DDST probe to a Static one. The amount of time spent executing instrumentation probes can be estimated from the number of probes hit times this cost. Table 5.9 shows the number of Static, DDST, and Static Agrawal probes executed on SPECjvm98. Alternatively, ρ can be used to determine if there have been enough dynamic probes saved by DDST to choose it over Static.

5.4.2 Agrawal Probe Reduction & Planning Cost

The model also needs to predict the improvement in dynamic probes as a result of applying Agrawal's algorithm. The last column of Table 5.9 shows α, the average reduction in dynamic probes via Agrawal's algorithms. The reduction varies significantly, from only 2.2% in *jack* to 26% in *jess*. On average, there is a 12.6% reduction.

Ideally, a model for computing α from a static property of a test region would be incorporated into the HST Planner. However, α does not correlate well with the number of nodes or edges in a test region, as is shown in Figure 5.9. To better see the trend, an outlier corresponding to a method that had 600 edges was removed from the data prior to generating the plot. Without a model to predict the value, the average, $\alpha = 12.6$, is a

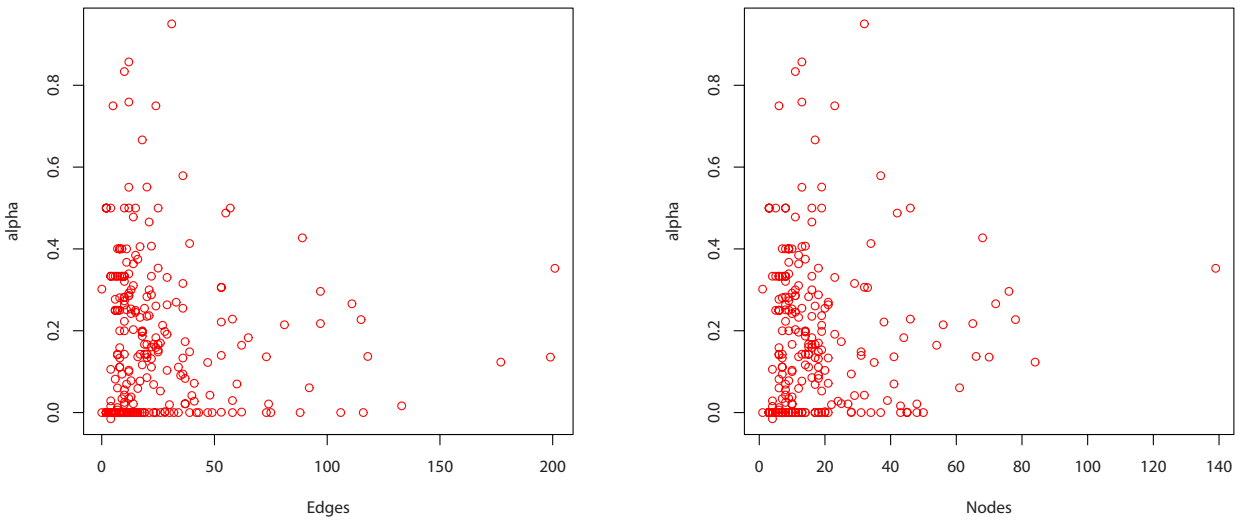

Figure 5.9: α does not correlate well with edges or nodes.

reasonable value to use.

Any reduction in probes comes at the increased cost of planning due to the amount of control-flow analysis that Agrawal's algorithm requires. Since the Agrawal planner is quadratic in the number of edges, a simple regression model can be created to predict the cost of planning from the observed costs. To determine the cost of planning, the cost of a baseline uninstrumented run was subtracted from a run of the program without inserting instrumentation. The resulting time of planning was fit using a regression model in the R software package [66]. The resulting formula is:

$$C_{Agrawal\ Plan} = 134967 \times Edges^2 + 12732371$$

The correlation is $R^2 = 0.9951$ and the result is in nanoseconds. The data and regression line are shown in Figure 5.10. This equation forms the implementation of the `AgrawalPlannerCost` method in the HST Planner shown in Algorithm 5.1.

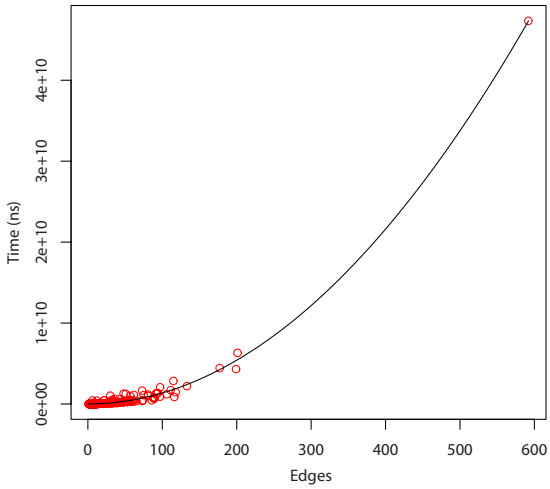

Figure 5.10: Regression curve for the cost of Agrawal planning.

5.5 EVALUATION

To determine the effectiveness of HST in improving the runtime performance of branch testing, I implemented the HST Planner and used the resulting HST Plan to drive Jazz to perform branch coverage testing on SPECjvm98. The experimental design for these results is identical to the one presented in Section 4.3 for DDST.

5.5.1 Overhead

Figure 5.11 shows the performance of HST on SPECjvm98. These benchmarks have three input sizes, 1, 10, and 100, with the largest being the default full run. To generate the HST Plan, each benchmark was run with the counting instrumentation probes to collect the number of dynamic probe executions for DDST and Static.

The HST Planner took this profile and used $\rho = 18.2$ for the relative cost of a DDST probe to a Static one and $\alpha = 12.6$ for the dynamic probe reduction from Agrawal. It produced a plan for each method based on the results of the model.

Figure 5.11 compares two runs of HST to Static and DDST, as well as against the

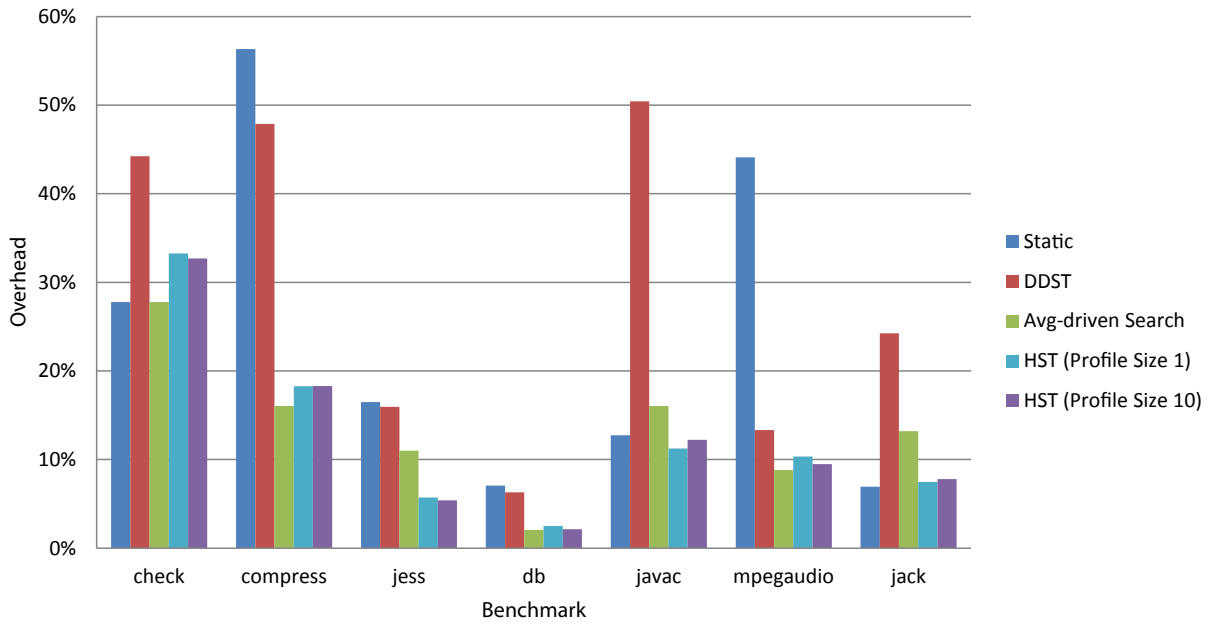

Figure 5.11: Results for a size 1 and size 10 profile run.

average-driven search. Uniform application of Agrawal is omitted since it never is the best technique. The two runs of HST are profiled on the size 1 and the size 10 inputs respectively, and the resulting HST Plan is run on the full size 100 input.

The most improvement from using HST can be seen in *jess*. This benchmark is an expert system that solves logic problems using rule-based inference. The HST Planner selects DDST for several methods that deal with rule lookup and filtering. These actions are frequently executed and loop-intensive. However, the source code commonly contains labeled break and continue statements that result in stranded blocks—where Static performs best. The mix of techniques allows each method to get the best-suited test technique.

In every benchmark except *check* and *jack*, HST is better than uniformly doing either Static or DDST. The short runtime of *check* and sensitive GC interaction make it a poor benchmark to judge HST on. The model does not account for GC effects. The result of average-driven search being the same as Static shows that choosing Static uniformly is

Table 5.10: Frequency of the test techniques chosen by HST.

	Size 1				Size 10			
	DDST	Static	Agrawal	Unseen	DDST	Static	Agrawal	Unseen
check	6	41	0	32	6	41	0	32
compress	10	9	0	23	10	9	0	23
jess	54	47	0	335	55	47	0	334
db	3	15	0	9	4	13	0	10
javac	30	125	0	587	175	190	0	377
mpegaudio	58	24	0	119	59	24	0	118
jack	87	58	0	121	97	48	0	121

best—any choice of DDST or Agrawal hurts overall performance. The result of HST on *jack* is only slightly worse than the uniform application of Static.

HST even performs better than average-driven search in most cases, the exceptions being *check* and *compress*. With *compress*, the overhead is only 2.2% over the average-driven search results. Considering the amount of computational effort necessary to perform the search, HST is a much better approach that gives results on par at worst, and frequently better than average-driven search.

5.5.2 HST Plan

To achieve the results of the previous section, the HST Planner chose a mix of test techniques to combine into a single test run. Table 5.10 shows a breakdown of those choices. The generated plans from both the size 1 and size 10 runs are analyzed. For the benchmarks *check* and *compress*, the plans are identical for both inputs. The biggest change is in *javac*, where the number of unseen methods during the profile run nearly halves. The remaining benchmarks have only a small variance between the two sizes.

The most noteworthy trend, however, is the lack of any choice of Agrawal's technique. As the results have shown, uniform application of Agrawal performs poorly due to the increased cost of planning. This is an inherent drawback to using a small profile input to do HST planning. Agrawal's algorithm must be applied to the code of the full method,

Table 5.11: Adding an individualized Agrawal reduction ratio.

	Size 1				Size 10			
	DDST	Static	Agrawal	Unseen	DDST	Static	Agrawal	Unseen
compress	10	8	1	23	10	7	2	23

but the smaller inputs make it unlikely to have a sufficient dynamic probe reduction to recoup the cost of planning.

5.5.3 Agrawal Probe Reduction (α)

There are two hypotheses for why the HST Planner did not select Agrawal. One possibility is that the Agrawal reduction, $\alpha = 12.6\%$, being the average across all methods in the benchmark suite, was inappropriate to use for all methods or all programs. The other factor is that the profile runs do not execute long enough for the reduction in dynamic instrumentation probe count to amortize the cost of planning with Agrawal's algorithm.

To test the impact of α on the test plans, a modified HST Planner was implemented that took an additional profile as input. This profile contains the dynamic instrumentation probe counts from Static Agrawal on a size 1 or size 10 profiling run, as was done for the DDST and Static profiles. The modified HST Planner uses these results to compute a per-method α, rather than using 12.6% uniformly.

The resulting test plans generated by the HST Planner are identical. In no case does profiling an individual method to compute α lead to the planner selecting Agrawal for that method.

This strongly suggests that the profile runs are just too short. To verify this, a run of the HST Planner with $\alpha = 100$ was performed. This value representsthe HST Planner predicting that there will be no probes executed at runtime. The HST Planner will then select Agrawal for any method where the cost of instrumentation with Static or DDST is higher than the cost of Agrawal planning.

Running the HST Planner with $\alpha = 100$, six of the seven benchmarks still do not

Table 5.12: Results of using $\alpha = 100$ on *compress*.

α	**Profile**	**Overhead**
12.6	Size 1 & Size 100	18.3%
100.0	Size 1	12.6%
100.0	Size 10	13.9%

change in the HST Plan for either the size 1 or the size 10 profile. The lone exception is *compress*. The new HST Plan statistics are shown in Table 5.11. The size 1 profile adds one use of Agrawal and the size 10 adds two uses. When the resulting HST Plan is run on the full size 100 input, the overhead is reduced as shown in Table 5.12.

The four to five percent reduction in overhead shows that the introduction of Agrawal into an HST Plan can considerably reduce the cost of branch coverage testing. However, the cost of gathering the Agrawal profile as input to the HST Planner may outweigh any benefit in practice.

Compared to the HST Plan for *compress* shown in Figure 5.8, the size 1 HST Plan switches the `Decompressor.decompress` method from Static to Agrawal. The size 10 plan changes `decompress` as well and additionally switches `Decompressor.getcode` from Static. The structure of `decompress` was described at the beginning of the chapter as one of the three methods shown in Figure 5.1. It suits Agrawal's technique perfectly: It contains a stranded block that is in the hot path of the code and is never covered, which is the situation where DDST performs poorly.

The `getcode` method is similar. It is called from the hot path of the decompression loop to look up codewords out of the compressed file. It contains a check for end-of-file (EOF) that is an If-Then stranded block that remains uncovered for every iteration until EOF is encountered; too late for DDST to perform well. However, changing `getcode` from Static to Agrawal causes a slight decrease in performance. This is unsurprising because there is not actually a 100% reduction in instrumentation probes executed at runtime.

Setting $\alpha = 100$ and having performance improve for *compress* strongly suggests that

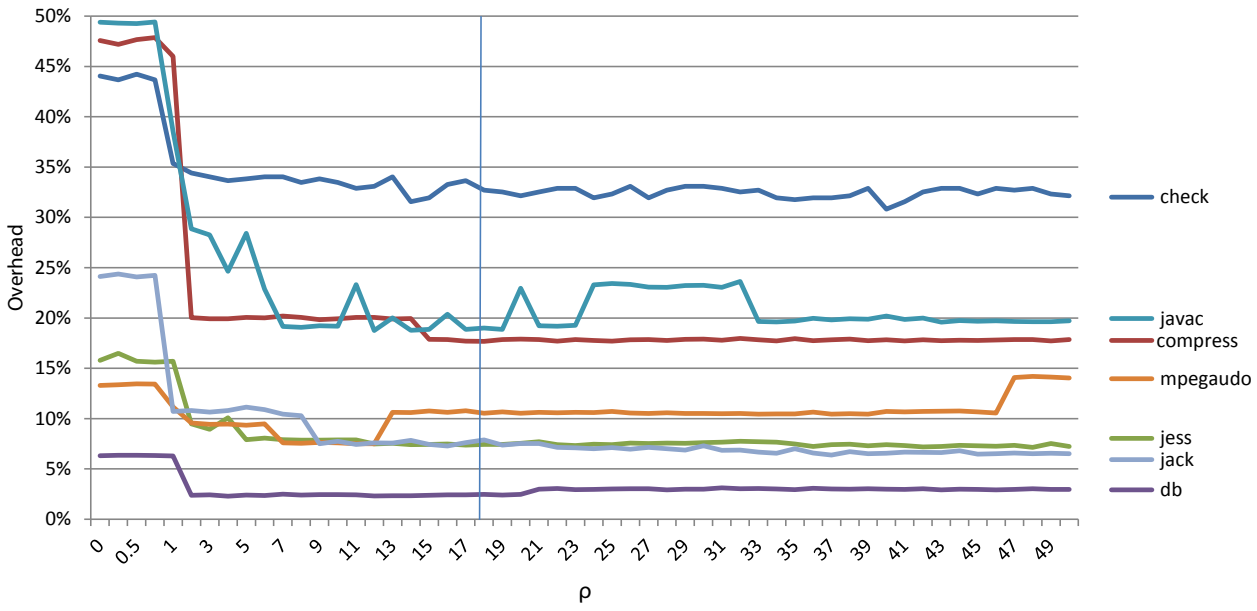

Figure 5.12: Sensitivity of ρ.

the short profile runs are the reason the HST Planner never choosing Agrawal. The only two methods selected are both part of the benchmark that has the highest iteration count where the benefit of probe reduction can overcome the cost of Agrawal planning.

5.5.4 Probe Cost

When instantiating the model, ρ was calculated to be 18.2 by measuring the ratio on each of the seven SPECjvm98 benchmarks and computing the average. To assess how good of a choice this value was, I did a sensitivity study that varied the value of ρ from 0 to 50. The results are shown in Figure 5.12. Note that the X-axis is not linearly scaled for values less than one.

The value $\rho = 0$ represents a completely free DDST instrumentation probe and the results correspond to running DDST on all methods as was shown in Table 5.1 and Figure 5.5. Unity represents where the cost of a static probe and a demand-driven probe are equal. This results in the HST Planner choosing whichever technique has the least

Table 5.13: HST Planner cost in seconds.

	Size 1			Size 10		
	Profile	HST Planner	Total	Profile	HST Planner	Total
check	3.03	0.00	3.03	3.04	0.00	3.04
compress	5.86	0.00	5.86	6.80	0.00	6.80
jess	3.84	0.00	3.84	4.84	0.00	4.84
db	2.13	0.00	2.13	3.14	0.00	3.14
javac	6.18	0.00	6.18	9.46	0.00	9.46
mpegaudio	2.95	0.00	2.95	7.16	0.00	7.16
jack	5.27	0.00	5.27	7.03	0.00	7.03

instrumentation probes executed at runtime. DDST will never execute more probes than Static as, in the worst case, the same basic blocks are instrumented. So any method where even a single demand-driven probe is removed, the HST Planner chooses DDST for that method. The same number of dynamic Static and DDST probe executions can result from an instrumented method being executed exactly once or because the entire method suffers from the stranded block problem with coverage never reaching 100%. The HST Planner algorithm chooses Static over DDST in the case of a tie.

The values greater than one represent the situation where a demand-driven probe is more expensive than a static probe. The trend stabilizes quickly for values greater than about 10. Nearly any of these values would serve as an adequate choice for ρ. The 18.2 value for ρ proves to be a good choice for the average relative cost of a DDST instrumentation probe to a Static probe.

5.5.5 HST Planner Cost

Unfortunately, the improvement shown in Figure 5.11 does not come for free. For each new HST Plan generated by the HST Planner, a profile containing the dynamic instrumentation probe counts for DDST and Static (essentially a basic block profile in traditional profiling situations) must be gathered. Although these profiles are collected on small input runs, the cost of planning and special counting instrumentation still incurs overhead.

Table 5.14: Number of runs necessary to amortize HST planning cost.

	Size 1			Size 10		
	DDST	Static	Agrawal	DDST	Static	Agrawal
check	16	—	1	15	—	1
compress	1	1	1	1	1	1
jess	2	2	1	2	2	1
db	2	2	1	3	3	2
javac	1	17	1	1	77	1
mpegaudio	6	1	1	10	2	1
jack	2	—	1	2	—	1

Table 5.13 shows the cost of collecting the profiles on the size 1 and size 10 inputs. For each of the two input sizes, the cost of planning is decomposed into two parts. For the *Profile* column, a run of Jazz is timed that performs two operations: Each basic block is instrumented to gather a frequency of execution and DDST is planned and executed.

The *HST Planner* column reports the time spent executing the HST Planner. There is no measurable delay.

Finally, the *Total* column shows the full cost of planning, which incorporates the costs of the other two columns. The time is reported in seconds. It ranges from 2.13 seconds on a size 1 run of *db* up to 9.46 seconds for the size 10 run of *javac*.

The cost of HST Planning is done before a full testing run and contributes to the overall time overhead for branch testing. For this work to pay off, the cost of planning should result in a net speed up when testing over an entire test suite. Table 5.14 shows the number of runs on similarly-sized test cases that would be necessary to completely amortize the cost of planning.

Again, there are two sets of columns corresponding to the size 1 and size 10 profiles. For each profile, Table 5.14 shows the number of runs of the full-sized (size 100) input necessary to completely amortize the cost of HST Planning compared to naïvely running each of the three individual test techniques. For example, using DDST on *check* requires 2.54 seconds whereas HST only needs 2.35 seconds, a savings of 0.19 seconds. However,

Table 5.15: HST Planner with profiled α (cost in seconds).

	Size 1			Size 10		
	Profile	HST Planner	Total	Profile	HST Planner	Total
check	8.37	0.02	8.39	8.40	0.02	8.42
compress	7.38	0.01	7.39	8.30	0.01	8.31
jess	18.08	0.04	18.12	18.98	0.04	19.02
db	3.83	0.01	3.84	4.84	0.01	4.85
javac	37.34	0.10	37.44	51.18	0.17	51.35
mpegaudio	6.61	0.02	6.63	12.51	0.02	12.53
jack	62.99	0.06	63.05	64.56	0.07	64.63

profiling and the HST Planner took a total of 3.03 seconds to execute. Thus, HST needs $\left\lceil \frac{3.03}{0.19} \right\rceil = 16$ runs to recoup the savings over just running DDST on all methods.

In four cases, HST did not improve on the uniform application of a single test technique. These are indicated in the table with an em dash (—). In each case, the HST Plan was unable to outperform the Static technique.

The number of runs necessary to amortize the cost of HST Planning ranged from 1 to 77, with 77 being a considerable outlier as the next highest value is only 17. For HST to be beneficial, this number of runs must be a realistic value when compared to real test suites. In a 2002 empirical study, Rothermel et al. explore the effectiveness of test suite reduction on the Siemens programs with test suite sizes ranging from 1052–5542 test cases [58]. These test suites can be reduced, but too small of a suite may lose the ability to detect faults. Practically, a test suite containing 77 test cases is reasonable to detect bugs in any large or important program.

When Agrawal is added to the profile run, the cost of HST Planning increases, as is shown in Table 5.15. The HST Planner now has a measurable overhead, but only takes a fraction of a second. The significant increase is the cost of profiling, which takes up to twelve times longer in the case of *jack*. The increase is a direct result of the additional step that is required in the profiling phase: Agrawal's technique must be run on each method's CFG and the result input into the HST Planner. No additional instrumentation

is necessary as the resulting Agrawal plan is a subset of the Static instrumentation probe locations, and the dynamic executions of those probes will be identical for both Agrawal and Static.

The maximum number of full-sized runs to amortize the HST Planner cost rises to 101 for *javac*. If profiling α as part of the HST Planner yielded any benefit, this is still probably a reasonable size for a test suite. However, the actual results show that there is no change in the HST Plans, and so HST Planning with a profiled α should not be done.

5.6 SUMMARY & CONCLUSIONS

The higher cost of DDST instrumentation probe versus a Static one is a result of multiple factors. DDST probes are larger in size and instruction count and do more work at runtime to insert and remove probes. Modifying a stream of instructions resulted in instruction cache and DTLB misses. The average probe in DDST is 18.2 times as costly as the average Static one. This cost is compounded by the stranded block problem, where the removal of probes is prevented.

The results show that no single test technique across multiple programs, or even multiple methods in the same program, performs best overall. A reasonable solution to this problem could match the strengths of each technique to the methods on which it will perform best. Unfortunately, it is generally the case that attempting to determine which technique to use *a priori* is impossible. Very similar methods may exhibit different runtime behaviors depending upon their inputs and result in different techniques being appropriate.

An exhaustive average-driven search showed that there frequently was a performance improvement from mixing multiple test techniques into a single run. However, the search time was prohibitively long and did not always outperform the uniform application of a single test technique.

An alternative, Profile-driven Hybrid Structural Testing (HST), was proposed based upon two premises: First, the test techniques can be modeled to a reasonable degree of

accuracy and, second, that a profiling run on a small input can realistically be used to predict the runtime behavior of a much larger test case. The models of the test techniques captured the reduction in dynamic instrumentation probes executed and the relative cost of the probes used for each technique. Additionally, the model incorporates the cost of planning when significant, as in the Static Agrawal branch coverage technique.

The results of this profile run are combined with the instantiated models to form the HST Planner, which produces a HST Plan. The SPECjvm98 benchmarks were profiled and HST Plans were created. The full runs of each benchmark were then instrumented using this plan. The results show that HST improves upon the naïve application of a single test technique an average of 48% versus Static and 56% versus DDST. The cost of profiling is small and can be amortized quickly as part of a test suite.

The use of HST resulted in a mix of test techniques that performed better than or on par with the average-driven search, but reduced the time spent determining that mix from over 100 hours to only a few seconds.

6.0 TEST SUITE-CENTRIC STRUCTURAL TESTING

THE PROFILE RUN for HST successfully allows the HST Planner to determine *a priori* whether to apply Static Branch or Demand Branch to a test region based solely on the dynamic instrumentation probe reduction. However, Static Agrawal was never chosen due to the high cost of planning. This is evidenced by manipulating α, the percentage reduction in dynamic probes that HST predicts. Even when α was set to 100%, meaning that no probes are predicted to be executed at runtime, only two methods in *compress* saved enough instrumentation probe executions to amortize the cost of Agrawal planning.

HST's performance, even without Agrawal, was overall better than the uniform application of any single test technique. However, the $\alpha = 100$ results show that the addition of Agrawal into the mix of techniques can improve testing performance significantly.

To choose Agrawal, the HST Planner must determine from the profile results that sufficient dynamic probe executions are eliminated to amortize Agrawal's planning costs. One way to obtain this is to use a longer-running profile input, but that is contrary to the goal of HST Planning to be quickly amortized. Alternatively, there may be a way to reduce the cost of Agrawal planning sufficiently that the HST Planner could be justified in choosing it.

Figure 6.1 shows the overhead that would result if Agrawal's planning was free. The obvious improvement over the planning-included Agrawal aside, this version of Agrawal finally shows the effect of reducing the number of instrumentation probe locations. It outperforms Static in every case. Additionally, it is the best technique in all cases except for *mpegaudio*, where DDST performs best.

Clearly then, Agrawal with improved planing shows promise as an individual technique and a choice in HST. Unfortunately, the algorithm does not lend itself to any

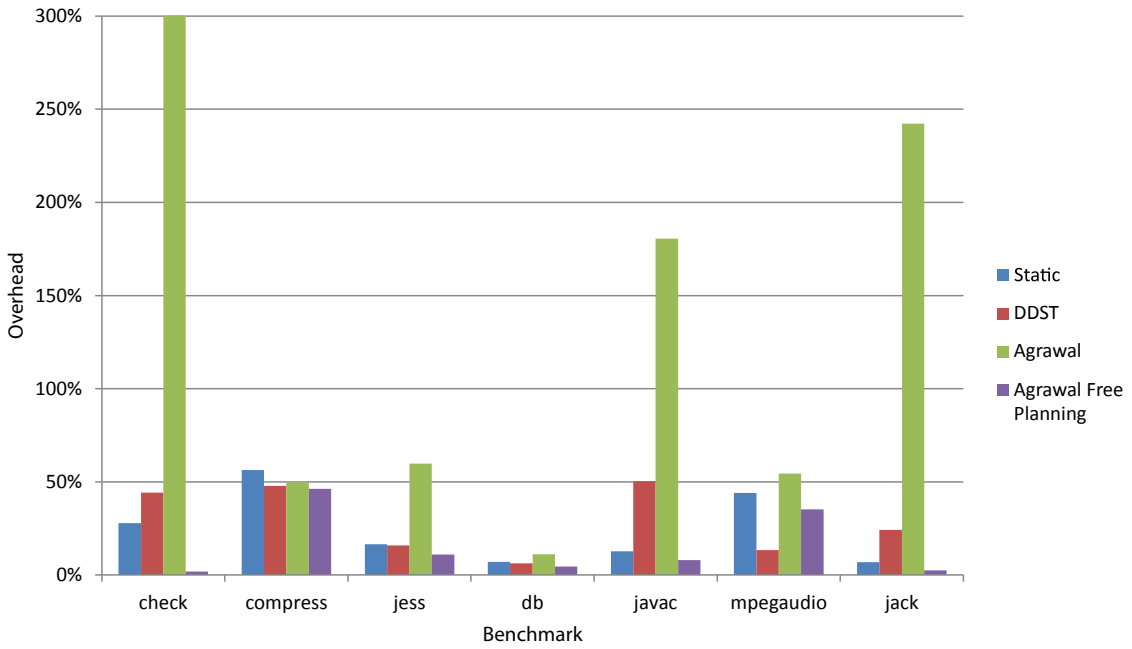

Figure 6.1: Overhead of Agrawal without the cost of planning.

significant improvement in asymptotic runtime. Instead, it may be possible to exploit the inherent iteration in the test process as was done with HST.

Agrawal in HST suffers from an additional problem beyond the small profile size. The other issue is that, while the input is scaled down, the planner does not get a reduced version the CFG of the test region. The profile run executed the full program code. Each test case, regardless of its size, is run on the same program code. It is only when the program is modified and a new round of testing begins, such as regression testing, that the code may be changed and require a new test plan. If during the first run of the program—such as HST's profile run or any test case from the suite—the results of Agrawal's algorithm were saved, subsequent program runs could reuse the plan without having to compute them anew. A sufficiently-sized test suite should once more allow for the cost of populating this test plan cache to be amortized.

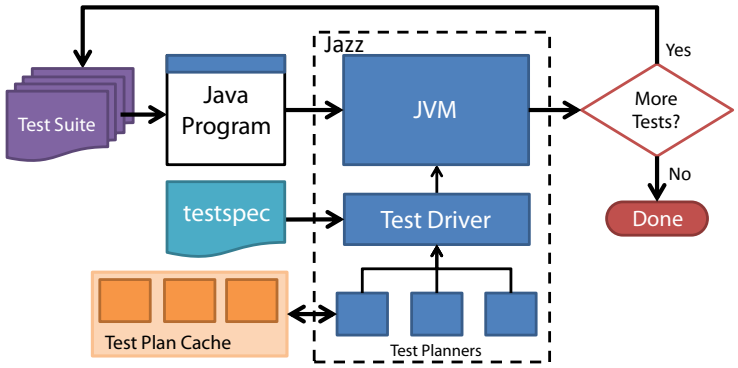

Figure 6.2: Cached test plans can be reused across the test cases in a suite.

6.1 TEST PLAN CACHING FOR STRUCTURAL TESTING

Figure 6.2 shows an overview of Jazz modified to support Test Plan Caching (TPC), the saving and loading of test plans. A test planner is modified to support loading a test plan from a file in addition to computing it at JIT compile-time. The saved test plans reside in a test plan cache.

When a test region is about to be instrumented, the test planner examines the cache to find if there is a saved plan for the given region. If there is not, the plan is computed and saved into the cache. If an appropriate plan exists, it is loaded and the planner uses the loaded plan to place instrumentation. The test planner must also allocate the appropriate data structures to record coverage and drive the runtime behavior of the instrumentation probes.

A "hit" in the test plan cache is possible even if the test region has been slightly altered. Planning depends upon the structure of the CFG of the region, and the addition or removal of statements that do not alter the control flow of a region will not result in a different test plan (changes to the structure may lead to incorrect coverage being recorded or inferred). This means that a test plan could be reused as long as the CFG is isomorphic to the one on which the saved test plan was generated. However, GRAPH ISOMORPHISM is in NP [27] and recomputing plans for any changed method may be faster.

6.2 IMPLEMENTING TEST PLAN CACHING IN JAZZ

Test plans in Jazz are represented by two Java data structures. First is a list of the seed set. The second is a table that holds a counter for each edge to cover as well as an identifier to determine which branch was taken at runtime. This table for Static Branch and Static Branch Agrawal was shown in Figure 3.7. Each of these data structures in Jazz supports Java's object persistence library, referred to as *serialization* [64].

The first version of TPC was built for Static Agrawal using Java's intrinsic serialization. Agrawal was chosen because it presents the best opportunity for TPC to make a difference. The test plan for the reduced set of edges to instrument was saved to the cache and support was added to load saved plans on subsequent runs of Jazz, eliminating the need to rerun the Agrawal planner.

This new version of Jazz was run on SPECjvm98 to evaluate its overhead. A preliminary run of all seven benchmarks on the full size 100 inputs was made to allow for the cache to be populated with test plans. With the cache filled, the benchmarks were rerun according to the experimental design detailed in Section 4.3.

The results are shown in Figure 6.3. In each case, TPC has considerably reduced the cost of planning to where a cached Agrawal is competitive with the Static technique. In four of the benchmarks, *compress*, *jess*, *db*, and *mpegaudio*, TPC outperforms Static. The remaining three benchmarks have a large number of methods or a short running time. Higher overhead in these benchmarks indicates that Java's deserialization is expensive, and comparison to the "free planning" Agrawal shows that there is significant room for improvement.

6.2.1 Saving Test Plans with *planspec*

To avoid the overhead of using Java's serialization, a custom language for expressing saved test plans was developed called *planspec*. The planspec language is designed to express the parts of a test plan sufficiently well to reconstruct the test plan, but must avoid storing any runtime-dependent values such as addresses. A planspec plan consists

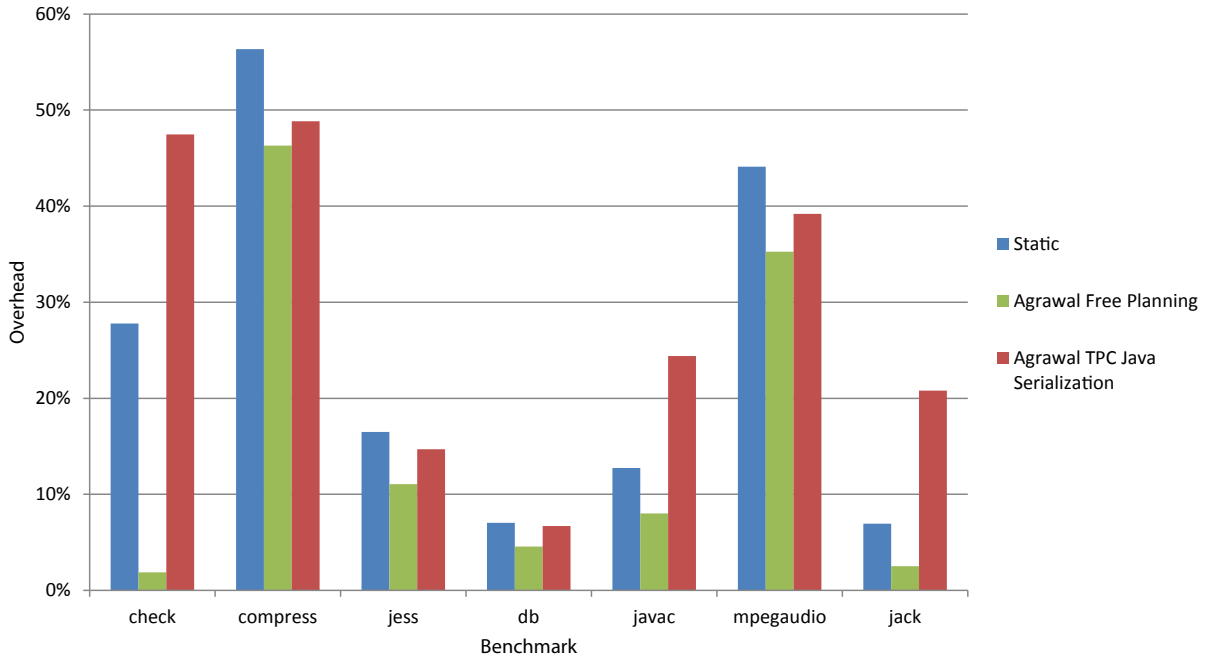

Figure 6.3: Loading a saved plan for Agrawal using Java's serialization.

of two parts. The first part saves the set of seed probes for the test region and the second part encodes the steps necessary to recreate a test plan in memory.

Table 6.1 shows the commands of the planspec langauge. The *SeedSet* command introduces an arbitrary-sized list of the locations that must be seeded. For Agrawal, the seed set is the same as the location of all instrumentation probes necessary to record coverage according to the algorithm. The *Table* command indicates the size of the test plan table. The table will have *probes* number of rows. Each edge to be tested has an identifier and a counter in which to record coverage at runtime. This identifier is the value that the predecessor probe has placed in the previously hit block variable and uniquely identifies the control flow edge taken to a given instrumentation probe.

The remaining two commands direct how to fill in the test plan table. The number of predecessors of each basic block is set using the *setNumPreds* command. This also allows the planspec loader to skip the appropriate space to accommodate the storage necessary for the identifier and counter storage. The *setPredAddr* command sets the identifier field

Table 6.1: Commands in *planspec*.

Command	Parameters			Description
SeedSet	...			Set of basic blocks that will be seeded with probes.
Table	*probes*	*edges*		Construct a table with *probes* rows and storage for *edges* address and coverage counters.
setNumPreds	*row*	*numPreds*		Define the size of row *row* to be *numPreds*.
setPredAddr	*row*	*pred*	*index*	Set Table[*row*][*pred*] to point to the address of Table[*index*].

so that the control flow edge to the current probe can be determined at runtime.

An example of planspec for `Compressor.compress` from *compress* is shown in Figure 6.4. There are 20 basic blocks to instrument, including the sentinel exit block labeled as -2. These 20 blocks are connected by 15 edges. The method has an additional 16 edges that can be omitted from the planspec because Agrawal's technique is able to infer their coverage rather than needing to record them directly.

6.2.2 Evaluation

The loading of saved test plans using planspec (TPC-planspec) is compared to Java serialization (TPC-serialization) and the original three branch coverage techniques in Figure 6.5. TPC-planspec has lower overhead in all seven benchmarks compared to TPC-serialization. However, in no case has TPC-planspec outperformed Static when TPC-serialization did not. TPC is the best technique for *jess* and ties for best with DDST in *compress* and *db*. However, once again there is no one technique that is consistently the best across all of the programs.

In *jack*, *javac*, and *check*, TPC-planspec significantly reduces the cost of plan loading over TPC-serialization. Since *check* runs so quickly, the difference is likely due to the initialization cost of using Java's serialization library. The other two programs have the most edges and nodes in their CFG, and the resulting plans are large. The size of the

```
SeedSet 280 136 276 227 179 131 269 28 118 258 163 22 158 61 197 149
146 192 -2 283
Table   20      15
setNumPreds     0       3
setNumPreds     1       0
setNumPreds     2       0
setNumPreds     3       1
setNumPreds     4       0
setNumPreds     5       0
setNumPreds     6       0
setNumPreds     7       0
setNumPreds     8       0
setNumPreds     9       0
setNumPreds     10      1
setNumPreds     11      1
setNumPreds     12      1
setNumPreds     13      2
setNumPreds     14      2
setNumPreds     15      3
setNumPreds     16      0
setNumPreds     17      0
setNumPreds     18      1
setNumPreds     19      0
setPredAddr     0       0       6
setPredAddr     0       1       9
setPredAddr     0       2       2
setPredAddr     3       0       14
setPredAddr     10      0       15
setPredAddr     11      0       7
setPredAddr     12      0       15
setPredAddr     13      0       4
setPredAddr     13      1       8
setPredAddr     14      0       5
setPredAddr     14      1       17
setPredAddr     15      0       17
setPredAddr     15      1       16
setPredAddr     15      2       1
setPredAddr     18      0       19
```

Figure 6.4: Saved *planspec* for `Compressor.compress` in *compress*.

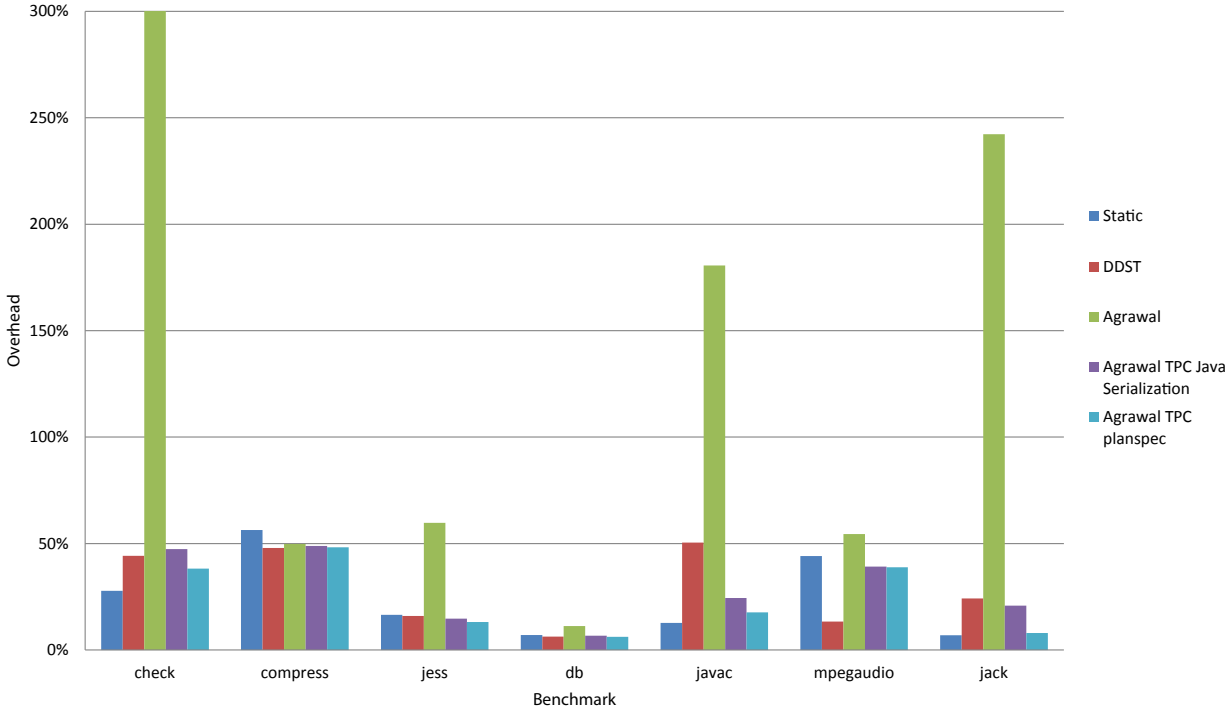

Figure 6.5: Loading a saved plan for Agrawal.

Table 6.2: Space necessary for planspec plans.

	Plan Size
check	19.3 KB
compress	5.2 KB
jess	39.4 KB
db	7.1 KB
javac	143.2 KB
mpegaudio	19.4 KB
jack	63.3 KB
Total:	*297.0 KB*

```
1: if N_static > ρ × N_demand then
2:   do_demand(m);
3: else
4:   do_tpc_agrawal(m);
5: end if
```

Algorithm 6.1: HST Planner with TPC Agrawal

saved planspec for each program is shown in Table 6.2. These results demonstrate that planspec should be used over Java's serialization.

Overall, the amount of space necessary for the plan cache is small. As was shown in Figure 6.4, planspec is text-based and human-readable. If space is a concern, a more compact representation could be used.

6.3 INCORPORATING TEST PLAN CACHING INTO HST

Agrawal was never chosen by the HST Planner due to its high cost and the short-lived profiles. However, TPC reduced the cost of planning to be on par with Static, and so caching Agrawal plans could allow for the test technique to be chosen in an HST Plan.

If TPC has truly made loading a saved Agrawal planspec as inexpensive as planning Static, then a new model for Agrawal could be constructed where the cost of Agrawal planning is zero as it was for Static and DDST. With no planning cost, the choice among the three techniques comes down to dynamic probe executions. DDST has the more expensive instrumentation probes, but Static and Agrawal share the same probe. Since Agrawal's algorithm finds a subset of edges in the test region, the number of dynamic probe executions for Agrawal can never exceed the total from Static. It can only be the same or less.

From this observation, it follows that the HST Planner should only choose between DDST and Agrawal with TPC and Static should be eliminated. The modified HST Planner

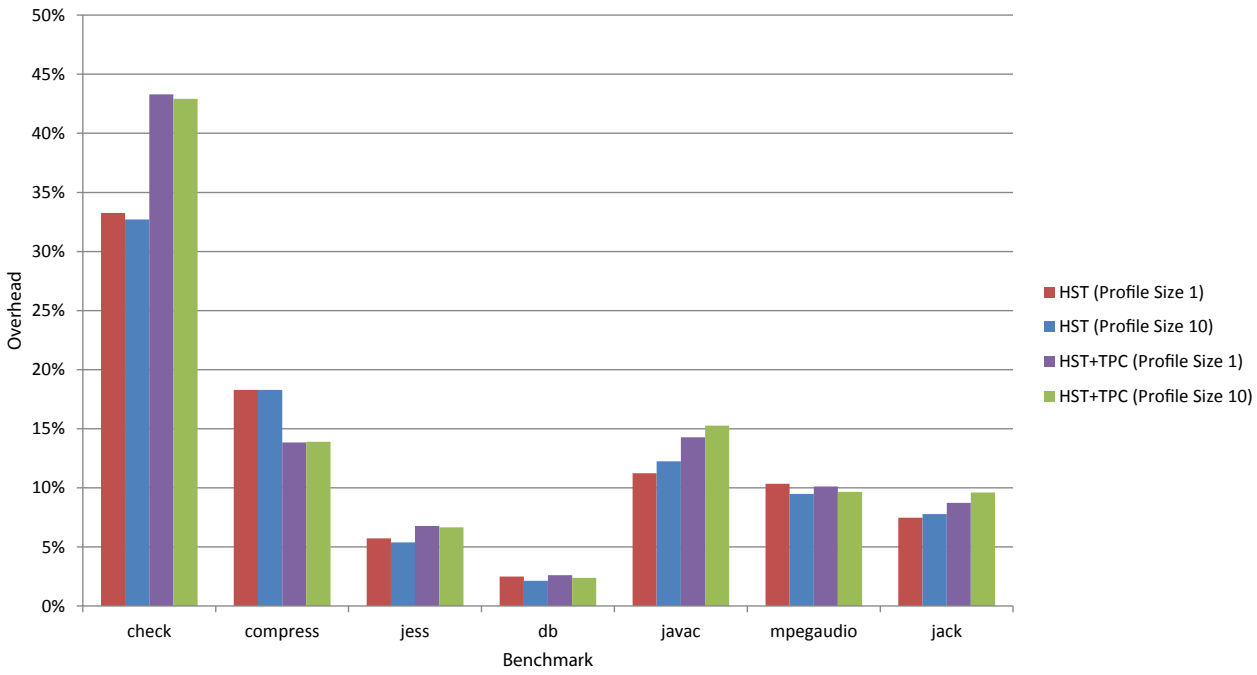

Figure 6.6: HST with Agrawal-TPC instead of Static for profiled methods.

incorporating TPC is shown in Algorithm 6.1. Line 1 still computes the tipping point between the more expensive DDST probes and how many dynamic executions were saved. If that balance falls in DDST's favor, it is chosen. However, if the original HST Planner would have chosen Static, TPC Agrawal is selected instead.

Static still retains a function in the generated HST Plan. For any method that is not encountered in the profile run, the HST Plan picks Static over Agrawal due to the cost of planning when no saved plan exists.

Figure 6.6 shows the result of this modified HST Planner. As with the $\alpha = 100$ experiment, *compress* again benefited from the inclusion of Agrawal. However, in five of the other six programs, performance degraded compared to the original HST Plan. For *mpegaudio*, the results are mixed. For the size 1 input, the HST+TPC Plan is very slightly better, but for size 10, the reverse is true.

Both *compress* and *mpegaudio* are loop-intensive programs and Agrawal's reduction can benefit a test region in a high trip count loop with an uncovered stranded block. The

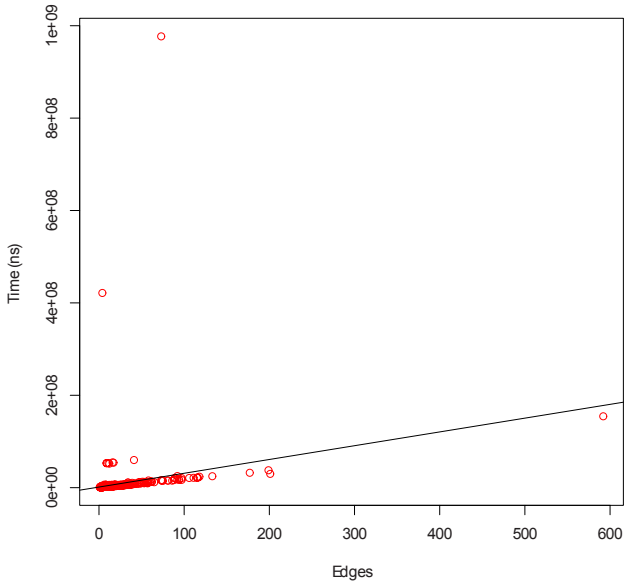

Figure 6.7: Regression line for the cost of loading an Agrawal plan in planspec.

other five programs are being delayed by TPC. Since the probes are the same cost and there are fewer probe locations that translate into a reduced dynamic execution cost, this overhead results from the loading of the test plan.

To address this, the cost of planning Agrawal needs to be brought back into the model as the cost of loading the planspec for a method.

6.3.1 Modeling planspec Loading

To model the cost of TPC plan loading, a procedure is used similar to the one in Section 5.4.2 that determined the cost model for Agrawal test planning. First, the test plan cache was populated with the planspec entries necessary for a full run of the benchmarks. The small amount of space necessary for the cache is reasonable enough for it to reside in RAM, which gives better and more predictable performance than plans stored on disk.

Jazz with TPC loading was run on the seven benchmark programs and the amount of time spent loading the planspec and creating the necessary runtime data structures was recorded. The data was loaded into the R statistical programming environment

```
1: if N_static > ρ × N_demand then
2:   do_demand(m);
3: else if α × N_static × C_static > AgrawalTPCCost(m) then
4:   do_agrawal(m);
5: else
6:   do_static(m);
7: end if
```

Algorithm 6.2: HST+TPC Planner with modeled planspec loading.

Table 6.3: Plan for *compress* under HST+TPC modeled planspec with $\alpha = 12.6$.

	Size 1				**Size 10**			
	DDST	Static	Agrawal	Unseen	DDST	Static	Agrawal	Unseen
compress	10	7	2	23	10	7	2	23

and regression was performed. The resulting scatterplot and regression line is shown in Figure 6.7.

Unlike with planning, loading planspec is linear in the number of edges. The formula for the model is:

$$C_{Agrawal\ TPC} = 298324 \times Edges + 1338020$$

The correlation is only $R^2 = 0.05199$ due to the large outliers. However, the apparent quality of fit is sufficient, if a slight overestimate of the true cost in the majority of cases.

With the model, the HST Planner now uses Algorithm 6.2 in a similar fashion to the original HST Planner.

6.3.2 Evaluation

The HST Planner with TPC-planspec and modeled cost of loading (HST+TPC+model) was run and HST Plans were generated. The Agrawal reduction ratio, α, was set to the average 12.6% value and the cost of loading a planspec plan was predicted according to

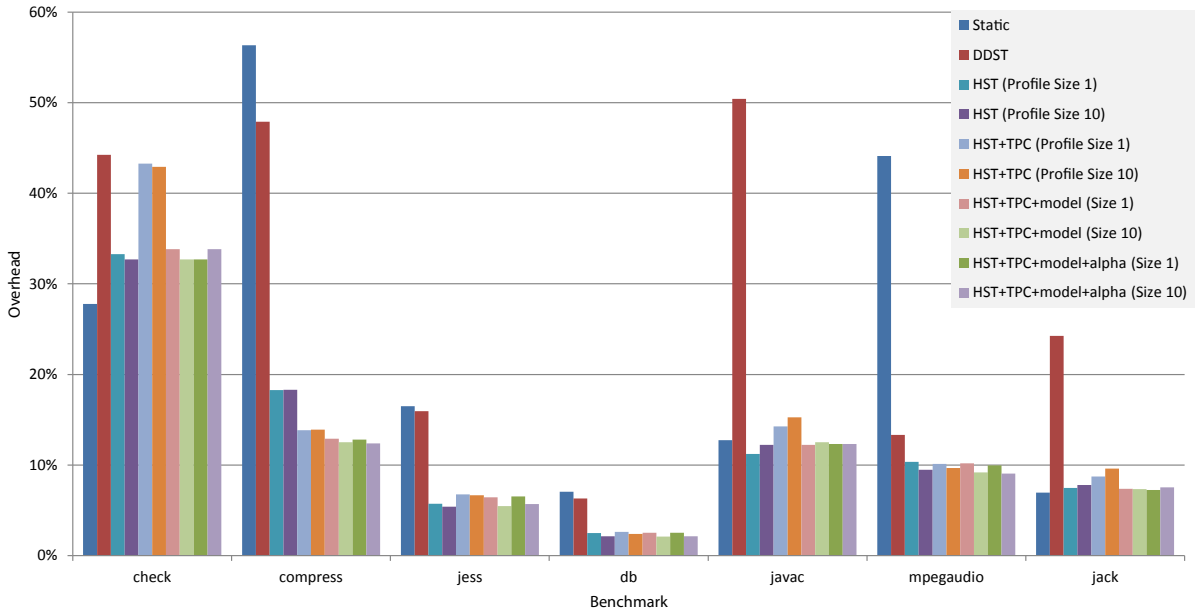

Figure 6.8: Overhead using HST+TPC with planspec loading modeled.

the model described in the previous section. In six of the seven SPECjvm98 programs, the HST+TPC+model plan was identical to the HST Plans detailed in Table 5.10 of Section 5.5.2. Once again, the only exception was *compress*.

In *compress*, two methods were selected for Agrawal TPC-planspec instead of Static: `Decompressor.decompress` and `Decompressor.getcode`. These were the two methods chosen by the original HST Planner when $\alpha = 100$.

Figure 6.8 shows the overhead when the resulting plans are executed on the full (size 100) inputs. For *compress*, HST+TPC+model reduces overhead with the inclusion of the two Agrawal-instrumented methods. The other methods are unchanged over the results of the original HST.

6.3.2.1 Profiled α For TPC, a per-method profiled α can be obtained with no additional overhead due to the profile runs performing Agrawal planning to pre-populate the test plan cache.

Table 6.4: Number of runs necessary to amortize HST+TPC+model+alpha planning.

	Size 1			Size 10		
	DDST	Static	Agrawal	DDST	Static	Agrawal
check	42	—	2	46	—	2
compress	1	1	1	2	1	1
jess	8	8	2	8	8	2
db	4	3	2	4	4	2
javac	4	377	1	6	518	2
mpegaudio	11	2	1	17	3	2
jack	16	—	2	17	—	2

The HST+TPC+model Planner was run with a profiled α and new plans were generated. Once again, a plan different from HST was generated only for *compress*. The only difference was that `Decompressor.decompress` was selected for Agrawal TPC-planspec. The other method selected for Agrawal with $\alpha = 12.6$, `Decompressor.getcode`, reverted to Static instrumentation.

The results from running the planner with a profiled α are shown in Figure 6.8 labeled as HST+TPC+model+alpha. For *compress*, the profile run on the size 10 input is now the best overall technique for branch coverage. The other benchmarks remain unchanged over HST.

Saving the Agrawal plans increases the cost of the profiling phase. With only *compress* showing a performance gain over the original HST, this extra cost of saving plans, plus the mandatory run of the Agrawal planner, translates into more full-sized test case runs necessary to amortize the cost of planning. The number of runs of the size 100 inputs needed is shown in Table 6.4.

For six of the seven benchmarks, the number of executions remains reasonable. Any test suite is likely to have sufficient test cases to cover the cost of profiling. However, on *javac*, the improvement over Static is so slim that the number of runs has increased to 317 when a size 1 run is profiled and to 518 with size 10 input. This is a direct result of the cost of saving the plans as *javac* has the most methods (742) by almost a factor of two over

the next largest benchmark. Even using the RAM disk, the cost of file I/O in Java is high.

6.4 SUMMARY & CONCLUSIONS

This chapter sought to address two issues. First, could the inherent repetition in testing an entire suite's worth of test cases be exploited to improve testing performance, and second, could Agrawal's technique be made less expensive enough to be chosen as part of an HST Plan.

A suite of test cases are all run on the same binary in a given testing run. Thus, any planning done for a region on the first test case will still be valid on the last test case. A test plan is dependent only on the structure of a method and thus can be cached for subsequent uses after it is initially computed.

The most promising technique to apply Test Plan Caching (TPC) to is Agrawal as the plans were by far the most expensive to compute. Two techniques for saving a test plan were explored: Java's serialization library and planspec, a new language for persisting test plans. Experiments showed that Java's deserialization was more expensive than loading a planspec plan.

With planspec loading and saving implemented in Jazz, HST was revisited to attempt to incorporate Agrawal. Since Agrawal's algorithm will never result in more dynamic probe executions that Static, and since they share the same probe, it was theorized that Agrawal with TPC could simply replace Static for any method seen in the profile run where DDST was predicted to be slower. Static would still be used on unseen methods due to the high cost of planning at runtime for Agrawal. However, this modified HST did not perform well. The cost of loading a planspec plan was still more expensive than computing Static's test plan.

A model for the cost of loading a planspec plan was developed and incorporated back into HST as a replacement for the cost of Agrawal Planning. New HST Plans were generated and only two methods in *compress* were chosen to be tested with Agrawal's technique. These methods were the same two selected by HST when $\alpha = 100$ was tested.

HST with TPC and modeled planspec loading results in a reduced runtime on *compress* and the same overheads as HST in the other six benchmarks.

Since the profiling phase must now pre-populate the test plan cache with Agrawal plans, the planner can profile individual method α values with no additional overhead. This results in one less selection of Agrawal for *compress* but results in a slight improvement over using $\alpha = 12.8$. Accordingly, HST+TPC+model+alpha replaces the original HST as the best technique for low-overhead branch testing.

With the cost of the profiling phase now including Agrawal and planspec saving, the number of full-sized test cases necessary to amortize planning is increased. In six of the seven benchmarks, the number of runs is small and any test suite is likely to contain sufficient test cases. For *javac*, however, the improvement over Static is small and the number of methods that must have plans saved is high. However, as was discussed in Section 5.5.5, real-world test suites may easily contain more than enough test cases to result in a net improvement.

7.0 CONCLUSION AND FUTURE WORK

SOFTWARE TESTING is a fundamental and useful part of the software development process. Determining the adequacy of a set of test cases that comprise a test suite is useful to understanding how thoroughly the cases exercise the code and expose bugs. A common way of determining the quality of a test suite is by coverage. Coverage reports for some structure such as a statement (node) or branch if it has been executed on a particular run of a program.

Traditional tools for collecting coverage information depend upon compiler or load-time insertion of instrumentation that remains throughout the entire execution of a program, incurring unnecessary overhead after coverage has been recorded.

This dissertation presented novel frameworks and techniques that allow for more efficient collection of branch and node coverage information. Jazz provided a framework on which to build and research branch and node coverage techniques, including Demand-driven Structural Testing (DDST), where transient instrumentation probes are inserted and removed during execution. DDST was the best technique for node coverage testing, being 19.7% faster than Static.

However, no single test technique is the best over all programs, or even all methods in a program. To address this, Profile-driven Hybrid Structural Testing (HST) was developed to match the best test technique to a particular method based upon a small profile run. The combination of multiple test techniques into a single run reduced the overhead of testing compared to a single test technique by an average of 48% versus Static and 56% versus DDST. Planning the combination of techniques to use took only seconds compared to over 100 computer-hours spent doing average-driven search.

HST never chose the use of Agrawal's static probe minimization algorithm due to

the high cost of planning. Inherent in the testing process is the repetition of running a program on each test case in a test suite. This repetition tests the same code, and thus the planning information, which only depends on the structure of the code, can be reused. Test Plan Caching (TPC) exploits this repetition and saves test plans for re-executions of the program. TPC enables Agrawal's reduction to be beneficial for long running code with uncovered stranded blocks such as in the *compress* benchmark.

In some cases, collecting the profile and pre-populating the test plan cache can require a large test suite with many re-executions. However, HST and TPC both can achieve enough of a reduction in collecting branch coverage to make them a useful part of the testing repertoire.

7.1 THREATS TO VALIDITY

The techniques presented in this thesis serve as a blueprint for efficient branch and node coverage testing that is generally applicable to a wide variety of architectures, programming languages, and runtime environments. Jazz's implementation specifically targets a Java JVM with JIT compiler executing on an Intel x86 processor. The experiments were run on a single machine type so that comparisons between techniques could be legitimately made. However, the performance of Jazz and the test techniques of this thesis could be affected by different choices of architecture, language, runtime environment, and multithreaded applications.

Intel's processors efficiently support self-modifying code and are designed to perform well despite effects on the cache hierarchy. An early experiment on an AMD processor found that it had a more naïve cache management policy, and thus, a single DDST instrumentation probe was more costly. If architectural details make a single DDST probe more costly than the experiments in this dissertation determined, the tipping point for choosing DDST over Static or Static Agrawal would change. However, the sensitivity study presented in Section 5.5.4 shows that the precise value of ρ does not greatly affect the runtime performance, provided that the model accounts for a DDST probe being at

least twice as expensive as a Static one. This effect is due to the stranded block problem as DDST probes will likely either be removed on the first execution or will never be removed. As long as the architecture allows for self-modifying code, DDST will perform well on methods with early coverage and few uncovered stranded blocks.

Programming languages other than Java may present certain control flow structures that Java does not support, such as a computed gotos and their corresponding indirect branches. However, Java has labeled break and continue statements as well as exception handlers that allow complex control-flow structures. Nevertheless, a CFG like the one shown in Figure 4.1 is unlikely to be produced by a Java compiler from Java source code. There are other compilers for languages like Scala [51] that produce Java Bytecode and it is possible to write a program in Java Bytecode directly with a tool such as the Jasmine assembler [42]. Using Jasmine, a valid Java classfile was created with the CFG of Figure 4.1 and the stranded block analysis of the DDST test planner was successfully verified on it.

The choice of Java as the targeted language also comes with a runtime environment, the JVM. With the runtime present, test planning cost is exposed to the user much the same as with other runtime code transformations such as register allocation. A traditionally-compiled language such as C can incorporate the cost of planning as part of compilation, but with a runtime environment, a balance must be struck between the work done in planning versus the resulting improvement of that effort. The caching of test plans between runs is a compromise between the challenges of doing analysis that is visible to the user and the offline, compiler-based approach while still being able to perform instrumentation at the machine code level.

Finally, DDST and threading interact in a number of ways. The previously hit block variable of branch coverage testing must be stored as a thread local variable. Additionally, probe insertions and removals must be done atomically, without any other thread executing the instruction stream at the point of insertion or removal. This implies locking the instruction stream with synchronization primitives, resulting in a reduction in parallelism. Shared payloads must be ensured to be reentrant, especially for the stranded block payload with its complex state. Finally, the definition of coverage must be clarified.

Is the coverage of a branch in one thread sufficient or should that branch be covered in all applicable threads? This will impact if the probe location table is a global shared resource or a thread-local data structure. Bringing the techniques of this dissertation to multithreaded applications will require further consideration as part of future work.

7.2 SUMMARY OF CONTRIBUTIONS

This dissertation has presented a number of contributions to the challenges of efficient branch and node coverage.

1. The first research contribution is a framework for structural testing. The framework consists of two parts, a formalized notation that expresses the actions necessary for structural testing, and Jazz, a software framework that supports the convenient implementation and evaluation of various structural testing algorithms.
2. Jazz provides facilities for interfacing with a Java JIT-compiler and Virtual Machine, adding instrumentation, managing memory, and performing control flow analyses. I demonstrated the usefulness of Jazz by implementing a library of tests, including implementations of the traditional static, compiler inserted instrumentation probes for branch and node coverage. Additionally, Agrawal's algorithm for inferring coverage is implemented and applied to both branch and node coverage.
3. Jazz also supports a language, testspec, that allows for the specification of how and where to test. This allows multiple test techniques to be combined into a single testing run.
4. This dissertation details a novel approach for structural testing using demand-driven instrumentation (DDST). DDST uses instrumentation probes built from fast breakpoints, allowing runtime insertion and removal of instrumentation. Both branch and node coverage are implemented using DDST.

   For branch coverage, DDST places instrumentation probes in basic blocks to record edges at runtime. This dissertation details the algorithms and data structures necessary to plan where to insert probes as well as the actions they take at runtime. A formal

definition of the stranded block problem, where instrumentation probes may be removed too early is provided, as well as a general solution with an optimized special case.

5. Unfortunately, DDST did not always outperform the traditional static instrumentation. With no one technique uniformly the best across programs or methods, this dissertation provides Profile-driven Hybrid Structural Testing (HST), where a small profile input is run with DDST and Static instrumentation. The reduction in dynamic probe counts is used to predict which technique to use.

   A model predicting the costs and benefits of each test technique was developed and instantiated for Static Branch, Static Agrawal Branch, and DDST Branch. The high cost of planning in Agrawal's technique required development of an additional cost model predicting the time spent planning with his algorithm.

6. Finally, this dissertation presented a way to exploit the inherent repetition of the software testing process as each test case is provided to the same program. Test Plan Caching (TPC) allowed for the saving and loading of test plans to avoid recomputation on future runs.

   To quickly save and load plans, a new language, planspec, was developed. Saving and loading planspec plans proved to be better than directly serializing the internal data structures due to Java's serialization library's high cost.

7.3 FUTURE WORK

While this dissertation presents an end-to-end solution for the efficient development, specification, and usage of structural test techniques, there still remains many new directions to explore as future work.

For Jazz, future directions could include extending the test library to include data-flow coverage criteria such as definition-use coverage, where coverage records which definitions of a variable reached a use. These additional techniques would help extend the flexibility and generality of the framework beyond the current control-flow coverage

techniques. Jazz could also be ported to other JVMs to assess the low-level interfaces it proves.

A major avenue of research for DDST would be to discover an improved solution to the general stranded block problem. One possible approach would be to break up the general case into small "patterns" that have improved, specialized solutions, as was done for if-then stranded blocks. Eliminating the need for the anchor probes to remain until a stranded block is completely covered may allow for DDST to be a better general-purpose test technique and to be utilized more frequently in HST.

Other avenues of probe reduction could be explored as well, especially using runtime information as might be available in a JVM. For instance, Agrawal's algorithm often presents a set of nodes or edges where only one out of the set needs to be covered to infer the rest. Currently, which of the members of the set actually gets the instrumentation probe is chosen essentially at random. However, combined with a profiling run—such as the one for HST or the online profiling done for adaptive optimization—instrumentation probes could be placed in regard to the hotness of a node or edge, thereby moving statically-inserted instrumentation out of the critical path.

For HST, there may be other ways to predict the dynamic probe reduction than by doing a profile run. For instance, machine learning techniques could be used to predict the likely behavior of a test region. Challenges here revolve around what constitutes the appropriate feature set to describe and predict the runtime properties of the test region.

New directions for TPC could include extending planspec to describe additional test plans, such as for DDST. Another possibility would be to dynamically determine whether recomputing a plan or loading a saved one is faster. For small methods with a simple CFG, recomputation might be faster than the I/O and parsing necessary to load a plan.

APPENDIX

BRANCH COVERAGE EXAMPLE

This appendix shows an end-to-end use of Jazz to perform DDST Branch Coverage testing on *compress* for the `Compressor.compress` method. First, testspec is used to define the test to be run:

```
spec.benchmarks._201_compress.Compressor:compress:DEMAND_BRANCH
```

Next, Jikes RVM is run with the testspec input:

```
rvm -I:compress.testspec SpecApplication _201_compress
```

When the method is called for the first time, the JIT-compiler emits machine code and Jazz takes over to do test planning and instrumentation. The test planner for DDST Branch decides where to place probes. It identifies four stranded blocks, including one that is an if-then stranded block. The CFG for `compress` is shown in Figure A1. Each node in the CFG is labeled with its starting bytecode index. The three regular stranded blocks are blocks 139, 187, and 270, and are shaded light blue. The if-then stranded block is block 153 and is shaded yellow.

The test planner examines each of the 21 nodes and, for every node, adds an action to place a probe in the node's control-flow successors. The planner also allocates space for the 31 edge counters to record coverage. For each of the three regular stranded blocks, an additional counter is allocated to indicate the coverage of the stranded block. For these stranded blocks, additional actions are added to replace the anchor probes in the predecessors of the stranded block and all of the block's CFG siblings.

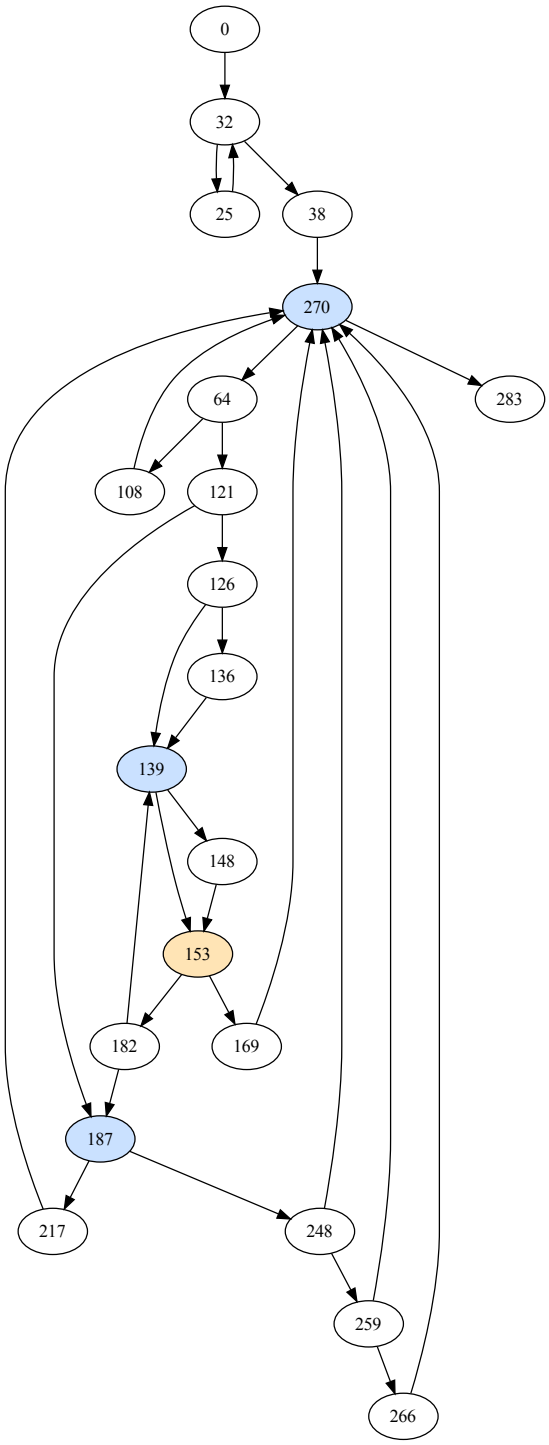

Figure A1: CFG of `Compressor.compress`.

```
Results for spec.benchmarks._201_compress.Compressor.compress():
806f5b0: | 7082706a | ff 74 24 20 ff | 2 | 5 | X ||
      708270bd | 207 | 806f518: 1 | 7082711e | 255 | 806f51c: 1 | 7082704f | 1e4 | 806f59c: 1 |
      7082704f | 1e4 | 806f598: 1 | 70826f9b | 35b | 806f598: 1 |
806f5f8: | 70826fca | ff 74 24 18 ff | 2 | 6 | X ||
      70826ff6 | 396 | 806f520: 1 | 70826fe6 | 3ba | 806f524: 1 | 7082704f | 1e4 | 806f59c: 1 |
      7082704f | 1e4 | 806f598: 1 | 70826fc4 | 256 | 806f59c: 1 | 70826fa7 | 2eb | 806f59c: 1 |
806f64c: | 70826fc4 | 6a 01 ff 44 24 | 1 | 2 | X ||
      70826fca | 237 | 806f528: 1 | 70826fa7 | 2eb | 806f59c: 1 |
806f670: | 7082704f | ff 34 24 ff ff | 2 | 4 | X ||
      70826fca | 237 | 806f530: 1 | 7082706a | 17e | 806f52c: 1 | 7082706a | 17e | 806f59c: 1 |
      70826fca | 237 | 806f598: 1 |
806f6ac: | 70826eb6 | 6a 08 ff 74 24 | 1 | 1 | X ||
      7082715c | 109 | 806f534: 1 |
806f6c4: | 7082715c | ff 74 24 20 58 | 2 | 9 | X ||
      70826ef6 | 44b | 806f538: 1 | 70827193 | 226 | 806f53c: 1 | 70827028 | 2b5 | 806f5a0: 0 |
      708270bd | 207 | 806f5a0: 0 | 70826eb6 | 396 | 806f5a0: 0 | 70826f74 | 3e6 | 806f5a0: 0 |
      7082714c | 15f | 806f5a0: 0 | 7082713a | 1ee | 806f5a0: 0 | 7082711e | 255 | 806f5a0: 0 |
806f73c: | 70826e97 | ff 74 24 1c ff | 2 | 2 | X ||
      70826eb6 | 396 | 806f540: 1 | 70826e82 | 48d | 806f544: 0 |
806f760: | 70826fa7 | ff 74 24 08 ff | 2 | 4 | X ||
      70826fca | 237 | 806f548: 1 | 70826fc4 | 256 | 806f54c: 1 | 70826fc4 | 256 | 806f59c: 1 |
      70826f9b | 35b | 806f598: 1 |
806f79c: | 7082714c | ff 74 24 20 ff | 1 | 2 | X ||
      7082715c | 109 | 806f550: 1 | 7082713a | 1ee | 806f5a0: 0 |
806f7c0: | 708270bd | ff 74 24 20 58 | 1 | 1 | X ||
      7082715c | 109 | 806f554: 1 |
806f7d8: | 70827028 | ff 74 24 20 58 | 1 | 1 | X ||
      7082715c | 109 | 806f558: 1 |
806f7f0: | 70826f9b | ff 34 24 58 ff | 2 | 3 | X ||
      70826fa7 | 2eb | 806f560: 1 | 7082706a | 17e | 806f55c: 1 | 70826fa7 | 2eb | 806f598: 1 |
806f820: | 70826e82 | ff 44 24 04 01 | 1 | 1 | X ||
      70826e97 | 3e7 | 806f564: 0 |
806f838: | 7082713a | ff 74 24 20 59 | 2 | 4 | X ||
      7082714c | 15f | 806f56c: 1 | 7082715c | 109 | 806f568: 0 | 7082714c | 15f | 806f5a0: 0 |
      7082711e | 255 | 806f5a0: 0 |
806f874: | 70826ef6 | ff 74 24 20 ff | 2 | 2 | X ||
      70826f9b | 35b | 806f570: 1 |70826f74 | 3e6 | 806f574: 1 |
806f898: | 70826f74 | ff 74 24 20 58 | 1 | 1 | X ||
      7082715c | 109 | 806f578: 1 |
806f8b0: | 7082711e | ff 74 24 20 59 | 2 | 3 | X ||
      7082715c | 109 | 806f57c: 1 | 7082713a | 1ee | 806f580: 1 | 7082713a | 1ee | 806f5a0: 0 |
806f8e0: | 70826ff6 | ff 74 24 20 58 | 2 | 4 | X ||
      7082704f | 1e4 | 806f584: 1 | 70827028 | 2b5 | 806f588: 1 | 70826fe6 | 3ba | 806f520: 1 |
      70826fca | 237 | 806f520: 1 |
806f91c: | 70826fe6 | ff 74 24 18 ff | 1 | 1 | X ||
      70826ff6 | 396 | 806f58c: 1 |
806f934: | 70827193 | ff 74 24 20 ff | 0 | 0 | X ||
806f940: | 70826e3f | 6a 00 ff 44 24 | 1 | 1 | X ||
      70826e97 | 3e7 | 806f590: 1 |
```

Figure A2: Coverage output from Jazz for DDST Branch.

Next, the actions and counters of the test plan are assembled into a PLT. A dump of the PLT is shown in Figure A2. The first number for each row is the address in memory of the PLT row. The next field is the address of the basic block corresponding to the PLT row. This is the location where the instrumentation probe has been placed. The five bytes that follow are the original machine instructions the probe overwrites and that will need to be replaced at runtime to remove the probe. Next are two one-byte numbers that represent the amount of actions for the probe to perform. The first is the number of coverage probes to place. The second is the total number of probes to place, including the anchor count. The "X" represents a single byte that pads the record for cache alignment.

For each of the probes to place—anchor and coverage—there are three entries. First is the address of where the probe should be placed. Next, the offset of the corresponding trampoline is stored so that the probe can be constructed at runtime. Finally, there is a pointer to the coverage counter for that edge (in a coverage successor) or the stranded block counter (for an anchor).

After the PLT is populated, assuming pre-seeding of the entire method, a fast breakpoint is placed at all 21 basic blocks. Jazz transfers control back to the JIT-compiler and the method is executed the first time. The program continues executing, collective coverage whenever `compress` is called. When the program terminates, Jazz is notified and displays the coverage information exactly as is shown in Figure A2. For each successor in the PLT, the coverage counter's value is shown as `1` for covered or `0` for uncovered.

The coverage information is post-processed to remove the artificial edges that the anchor probes have created. This leads to an annotated output as is shown in Figure A3. The CFG has the original edges (shown in black) labeled with the coverage results. Uncovered anchor "edges" are shown in red. Covered stranded blocks show their anchor edges in green. Yellow edges indicate a covered edge that is part of an if-then stranded block. Of the 31 true CFG edges, 28 are covered, resulting in 90.3% branch coverage on the method.

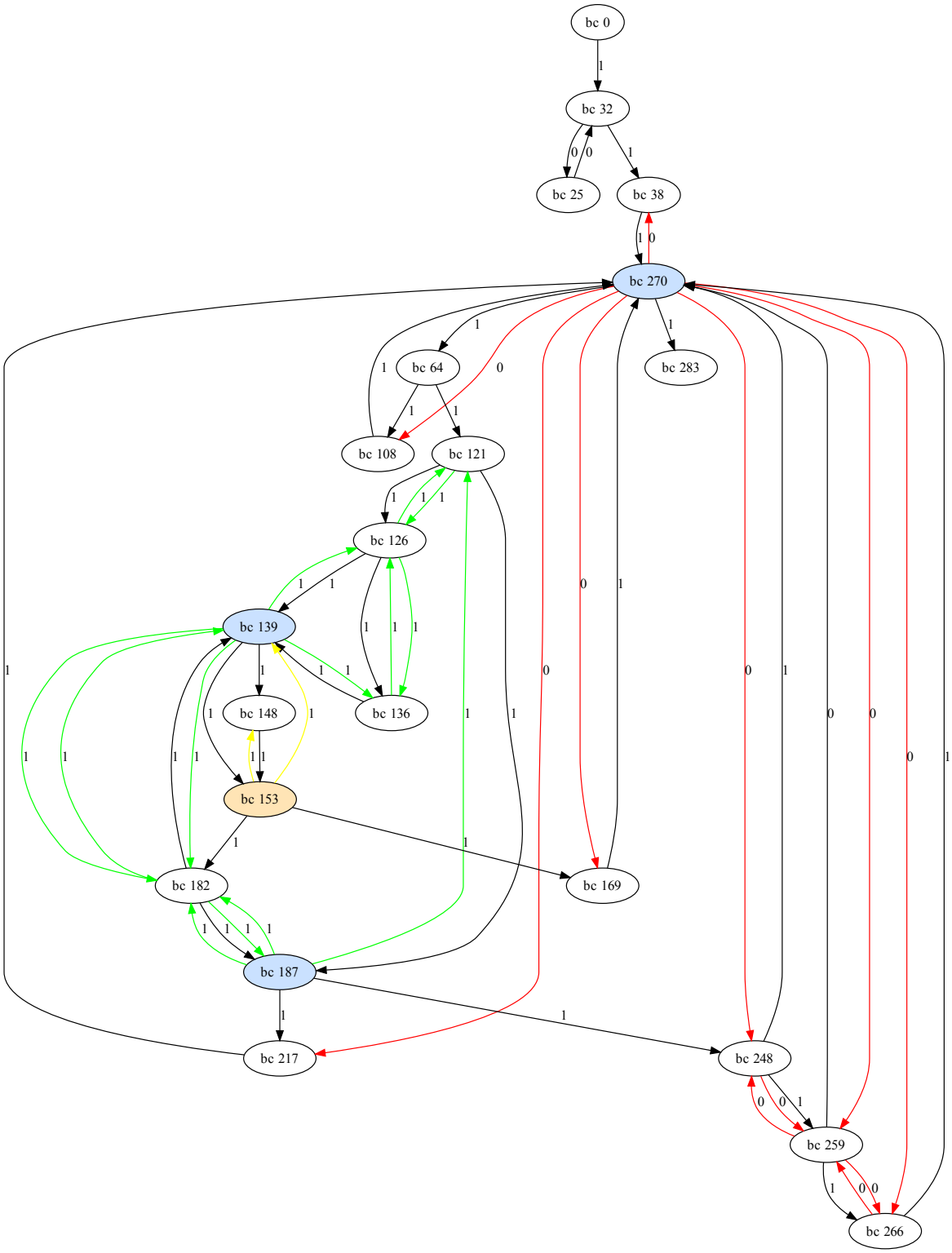

Figure A3: Results of DDST Branch Coverage on `Compressor.compress`.

BIBLIOGRAPHY

[1] The AspectJ Project [online]. Available from: http://www.eclipse.org/aspectj/.

[2] Bytecode Engineering Library [online]. Available from: http://jakarta.apache.org/bcel/.

[3] Clover [online]. Available from: http://www.cenqua.com/clover/.

[4] Cobertura [online]. Available from: http://cobertura.sourceforge.net/.

[5] Eclipse IDE [online]. Available from: http://www.eclipse.org/.

[6] EMMA [online]. Available from: http://emma.sourceforge.net/.

[7] JCover [online]. Available from: http://www.codework.com/JCover/.

[8] Jikes RVM [online]. Available from: http://jikesrvm.org/.

[9] Microsoft Phoenix Academic Program [online]. Available from: http://research.microsoft.com/phoenix/.

[10] Pin [online]. Available from: http://www.pintool.org/.

[11] Standard performance evaluation corporation [online]. Available from: http://www.spec.org/jvm98.

[12] Hiralal Agrawal. Dominators, super blocks, and program coverage. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–34, New York, NY, USA, 1994. ACM. doi:http://doi.acm.org/10.1145/174675.175935.

[13] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the jalapeño jvm. In *Conf. on Object Oriented Programming, Systems, Lang. and Applications*, 2000.

[14] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on*

Programming language design and implementation, pages 168–179, New York, NY, USA, 2001. ACM Press. Available from: http://doi.acm.org/10.1145/378795.378832.

[15] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. *SIGSOFT Softw. Eng. Notes*, 14(8):210–218, 1989. Available from: http://doi.acm.org/10.1145/75309.75332.

[16] Thomas Ball. On the limit of control flow analysis for regression test selection. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '98, pages 134–142, New York, NY, USA, 1998. ACM. Available from: http://doi.acm.org/10.1145/271771.271802.

[17] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, 1994. Available from: http://doi.acm.org/10.1145/183432.183527.

[18] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

[19] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings of the ACM 1999 conference on Java Grande*, JAVA '99, pages 129–141, New York, NY, USA, 1999. ACM. Available from: http://doi.acm.org/10.1145/304065.304113, doi:http://doi.acm.org/10.1145/304065.304113.

[20] Kalyan-Ram Chilakamarri and Sebastian Elbaum. Reducing coverage collection overhead with disposable instrumentation. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 233–244, Washington, DC, USA, 2004. IEEE Computer Society. Available from: http://dx.doi.org/10.1109/ISSRE.2004.32.

[21] R. Chilakamarri and S. Elbaum. Leveraging disposable instrumentation to reduce coverage collection overhead. *Journal of Software Testing, Reliability, and Verification*, 16(4):267–288, April 2006.

[22] B. R. Childers, M. L. Soffa, J. Beaver, L. Ber, K. Cammarata, T. Kane, J. Litman, and J. Misurda. Softtest: A framework for software testing of Java programs. In *Eclipse Technology Exchange Workshop during the ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'03)*, 2003.

[23] L.A. Clarke, A. Podgurski, D.J. Richardson, and S.J. Zeil. A formal evaluation of data flow path selection criteria. *Software Engineering, IEEE Transactions on*, 15(11):1318–1332, November 1989.

[24] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001. Available from: http://doi.acm.org/10.1145/383845.383853.

[25] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. Demystifying magic: high-level low-level programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 81–90, New York, NY, USA, 2009. ACM. Available from: http://doi.acm.org/10.1145/1508293.1508305.

[26] P.G. Frankl and E.J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, 1988. Available from: http://doi.ieeecomputersociety.org/10.1109/32.6194.

[27] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[28] Matthew Geller. Test data as an aid in proving program correctness. In *POPL '76: Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, pages 209–218, New York, NY, USA, 1976. ACM. Available from: http://doi.acm.org/10.1145/800168.811554.

[29] Mechelle Gittens, Keri Romanufa, David Godwin, and Jason Racicot. All code coverage is not created equal: a case study in prioritized code coverage. In *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, page 11, New York, NY, USA, 2006. ACM Press. Available from: http://doi.acm.org/10.1145/1188966.1188981.

[30] James Hall. Jaguar recalls cars after cruise control fault [online]. 2011. Available from: http://www.telegraph.co.uk/motoring/8841996/Jaguar-recalls-cars-after-cruise-control-fault.html.

[31] Jeffrey K. Hollingsworth, Barton P. Miller, Marcelo Goncalves, Oscar Naim, Zhichen Xu, and Ling Zheng. MDL: A language and compiler for dynamic program instrumentation. In *Conference on Parallel Architectures and Compilation Techniques (PACT)*, November 1997. Available from: http://www.cs.umd.edu/~hollings/papers/mdl.pdf.

[32] IBM. Rational purifyplus [online]. Available from: http://www.ibm.com/rational.

[33] IEEE. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, page 1, 1990. doi:10.1109/IEEESTD.1990.101064.

[34] Intel. *Intel® 64 and IA-32 Architectures Software Developers Manual*, volume 3B: System Programming Guide, Part 2. 2011.

[35] Clara Jaramillo. *Source level debugging techniques and tools for optimized code*. PhD thesis, University of Pittsburgh, 2000.

[36] J.A. Jones and M.J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *Software Engineering, IEEE Transactions on*, 29(3):195–209, March 2003. Available from: http://dx.doi.org/10.1109/TSE.2003.1183927.

[37] Cem Kaner, Jack Falk, and Hung Quoc Hguyen. *Testing Computer Software*. International Thompson Computer Press, London, UK, 1993.

[38] Gregory M. Kapfhammer. *The Computer Science and Engineering Handbook*, chapter Chapter 105: Software Testing. CRC Press, Boca Raton, FL, second edition, 2004. Available from: http://cs.allegheny.edu/~gkapfham/research/publish/software_testing_chapter.pdf.

[39] Gregory M. Kapfhammer. *A Comprehensive Framework for Testing Database-Centric Applications*. PhD thesis, University of Pittsburgh, Pittsburgh, Pennsylvania, 2007.

[40] Peter B. Kessler. Fast breakpoints: design and implementation. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 78–84, New York, NY, USA, 1990. ACM Press. Available from: http://doi.acm.org/10.1145/93542.93555.

[41] Naveen Kumar, Jonathan Misurda, Bruce R. Childers, and Mary Lou Soffa. Instrumentation in software dynamic translators for self-managed systems. In *WOSS'04*, 2004.

[42] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly Media, 1997.

[43] C.C. Michael, G. Mcgraw, and M.A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001. Available from: http://doi.ieeecomputersociety.org/10.1109/32.988709.

[44] Sun Microsystems. Java™ platform debugger architecture [online]. 2004. Available from: http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/.

[45] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *Computer*, 28:37–46, November 1995. Available from: http://dx.doi.org/10.1109/2.471178.

[46] J. Misurda, J. Clause, J. Reed, B. R. Childers, and M. L. Soffa. Jazz: A tool for demand-driven structural testing. In *International Conference on Compiler Construction*, 2005.

[47] Jonathan Misurda, Bruce R. Childers, and Mary Lou Soffa. Jazz2: A flexible and extensible framework for structural testing in a Java VM. In *9th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, August 2011.

[48] Jonathan Misurda, James A. Clause, Juliya L. Reed, Bruce R. Childers, and Mary Lou Soffa. Demand-driven structural testing with dynamic instrumentation. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 156–165, 2005. Available from: http://doi.acm.org/10.1145/1062455.1062496.

[49] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[50] National Institute of Standards and Technology. Software errors cost U.S. economy $59.5 billion annually [online]. 2002. Available from: http://www.nist.gov/director/planning/upload/report02-3.pdf.

[51] Martin Odersky. *The Scala Language Specification Version 2.9*. PROGRAMMING METHODS LABORATORY, EPFL, 2011. Available from: http://www.scala-lang.org/docu/files/ScalaReference.pdf.

[52] Jan K. Pachl. A notation for specifying test selection criteria. In *Proceedings of the IFIP WG6.1 Tenth International Symposium on Protocol Specification, Testing and Verification X*, pages 71–84, Amsterdam, The Netherlands, The Netherlands, 1990. North-Holland Publishing Co.

[53] Christina Pavlopoulou and Michal Young. Residual test coverage monitoring. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 277–284, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[54] Hridesh Rajan and Kevin Sullivan. Aspect language features for concern coverage profiling. In *Proceedings of the 4th international conference on Aspect-oriented software development*, AOSD '05, pages 181–191, New York, NY, USA, 2005. ACM. Available from: http://doi.acm.org/10.1145/1052898.1052914.

[55] Ian Rogers, Jisheng Zhao, and Ian Watson. Boot image layout for Jikes RVM. In *ICOOOLPS 2008: Third International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, 2008.

[56] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *Software Engineering, IEEE Transactions on*, 22(8):529–551, August 1996. Available from: http://doi.acm.org/10.1109/32.536955.

[57] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929 –948, October 2001. Available from: http://dx.doi.org/10.1109/32.962562.

[58] Gregg Rothermel, Mary Jean Harrold, Jeffery von Ronne, and Christie Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002. Available from: http://dx.doi.org/10.1002/stvr.256, doi:10.1002/stvr.256.

[59] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 179–, Washington, DC, USA, 1999. IEEE Computer Society. Available from: http://dx.doi.org/10.1109/ICSM.1999.792604.

[60] Raul Santelices and Mary Jean Harrold. Efficiently monitoring data-flow test coverage. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, Atlanta, GA, November 2007. Available from: http://www.cc.gatech.edu/aristotle/pdffiles/fp205-santelices.pdf.

[61] Alex Shye, Matthew Iyer, Vijay Janapa Reddi, and Daniel A. Connors. Code coverage testing using hardware performance monitoring support. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, AADEBUG'05, pages 159–163, New York, NY, USA, 2005. ACM. Available from: http://doi.acm.org/10.1145/1085130.1085151.

[62] Mary Lou Soffa, Kristen R. Walcott, and Jason Mars. Exploiting hardware advances for software testing and debugging: Nier track. In *Proceeding of the 33rd international conference on Software engineering*, ICSE '11, pages 888–891, New York, NY, USA, 2011. ACM. Available from: http://doi.acm.org/10.1145/1985793.1985935.

[63] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Trans. Softw. Eng.*, 22(11):777–793, 1996. Available from: http://dx.doi.org/10.1109/32.553698.

[64] Sun Microsystems, Inc. *Java^{TM} Object Serialization Specification*. Sun Microsystems, Inc., 1996–2001.

[65] Andrew S. Tanenbaum. In defense of program testing or correctness proofs considered harmful. *SIGPLAN Not.*, 11(5):64–68, 1976. Available from: http://doi.acm.org/10.1145/956003.956011.

[66] R Development Core Team. *R: A language and environment for statistical computing, reference index version 2.11.1*. Vienna, Austria, 2005. Available from: http://www.r-project.org/.

[67] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Efficient instrumentation for code coverage testing. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 86–96, New York, NY, USA, 2002. ACM Press. Available from: http://doi.acm.org/10.1145/566172.566186.

[68] Albert Tran, Michael Smith, and James Miller. A hardware-assisted tool for fast, full code coverage analysis. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, pages 321–322, Washington, DC, USA, 2008. IEEE Computer Society. Available from: http://dl.acm.org/citation.cfm?id=1474554.1475456.

[69] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997. Available from: http://doi.acm.org/10.1145/267580.267590.