Jazz2: A Flexible and Extensible Framework for Structural Testing in a Java VM

Jonathan Misurda
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
jmisurda@cs.pitt.edu

Bruce R. Childers
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
childers@cs.pitt.edu

Mary Lou Soffa
Department of Computer Science
The University of Virginia
Charlottesville, VA, 22904
soffa@virginia.edu

Abstract

This paper describes our experience designing and developing a framework for structural testing in a Java Virtual Machine, called Jazz2. We focus on the challenges of making this framework extensible and flexible. These challenges include developing the framework as part of a JVM that undergoes continual independent updates, the consequences of the JVM itself being written in Java, and how Jazz2 ties into existing code generation facilities to insert instrumentation for our static (compile-time insertion) and demand-driven (run-time insertion) testing techniques. We also address how best to add code and data to the managed environment that the JVM's garbage collector provides, including when to avoid it completely. Additionally, we provide suggestions for JVM designers and implementers on how to better support the addition of tools like Jazz2. We conclude by describing our implementation of Jazz2 with the IBM Jikes RVM. With this implementation, we evaluate the cost of determining how and where to test programs, memory overhead, and how those factors impact the number of garbage collections.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Testing tools; D.3.3 [*Programming Languages*]: Language Constructs and Features—Program instrumentation, run-time environments

General Terms Design, Experimentation, Measurement, Verification

Keywords Testing, Code Coverage, Structural Testing, Demand-Driven Instrumentation, Java Programming Language

1. Introduction

Testing the correctness of software is an increasingly important phase of software development. Countless hours are invested in creating suites of example inputs, running these suites, and collecting their results. The quality of the assembled test suite is assessed via a *coverage metric* [24], such as the percentage of statements executed or control-flow transfers taken in a given testing run. Typically, structural software tests *instrument* a program to determine which constructs (i.e., coverage) are executed by a test run. There are many important structural tests and instrumentation strategies [24]. We developed a framework, called Jazz2, that permits these approaches to be used, combined, and compared. The framework can be used to both test software and research and develop new structural test strategies. In this paper, we describe the challenges, solutions, and experiences in developing Jazz2 for Java and integrating it with a Java Virtual Machine (JVM).

Java presents a rich environment for efficient structural testing. 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. Indeed, a poor test strategy and inefficient tools can even delay the next major software release. 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.

A straightforward approach to structural testing could add instrumentation to a method's bytecode and leverage the JIT to convert the instrumentation bytecode into machine code. However, the instrumentation will remain in the generated machine code for the entire lifetime of the program unless an expensive full recompilation of a method is triggered (i.e., to remove the instrumentation). With a boolean property like coverage that in practice converges quickly, this approach incurs unnecessary overhead. We are also at the mercy of the quality of code generation and the effects of the Java runtime environment, including garbage collection.

We take an alternative approach in Jazz2 that inserts instrumentation directly into the generated machine code. This technique bypasses the JIT compiler. By directly instrumenting the generated code, Jazz2 offers low-level control over instrumentation behavior; the approach even permits instrumentation to be dynamically inserted and removed during program execution. Because instrumentation can be directly controlled, coverage testing can be made considerably more efficient, as past work has demonstrated [21].

Jazz2 was designed from its inception to be a flexible and extensible framework in which to research, develop, evaluate and use structural tests. Its extensibility is obtained through the ability to quickly and simply add new testing strategies either by modifying existing ones or building new ones with the set of support services Jazz2 provides. The framework is flexible through supporting 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.

Jazz2 permits structural testing that relies on dynamic techniques, such as our earlier work with the original version of Jazz [21]. Jazz was built on top of an early version of IBM's Jikes Research Virtual Machine (RVM) [13]. As the RVM underwent continual upgrades and improvements, Jazz was left behind without the bug fixes and design improvements of the newer RVM releases. Jazz2's design comes from the lessons we learned in implementing the original Jazz and reimplementing it to be more easily ported to new RVM versions.

This paper makes several contributions, including:

- The design of an extensible and flexible framework for structural testing of Java programs (Jazz2).
- A detailed description of Jazz2's implementation, including services for instrumentation and code generation, memory allocation, and test specification. We also describe how the framework is minimally integrated with the IBM Jikes RVM to ease independently upgrading the RVM and developing Jazz2.
- Case studies on how Jazz2 can be used to implement structural test techniques that rely upon different instrumentation styles. These tests are included in a test library as part of the framework. They can be extended to support new strategies.
- A thorough evaluation of the performance cost and memory overhead of doing structural testing in Jazz2, including its impact on garbage collection.
- We describe the rationale behind our implementation choices and the lessons learned through our experience with Jazz2. These lessons point to how a JVM can better support independent tools, such as Jazz2, that are built on top of it.

The rest of the paper is organized as follows. Section 2 describes Jazz2's design. Section 3 presents the framework's implementation approaches. Section 4 describes case studies on static and demand-driven branch testing to illustrate Jazz2. Section 5 reports on an evaluation of the framework, including performance and memory overhead and interaction with garbage collection. Finally, Sections 6 and 7 give related work and conclude the paper.

2. Framework Overview

Jazz2 is designed to let test engineers realize a specific structural test for Java through its extensible test library. The implementation of a testing strategy in Jazz2 is a *test planner* and the data structure that drives runtime instrumentation to record coverage is a *test plan*. An overview of Jazz2 is shown in Figure 1. Jazz2 integrates with a Java Virtual Machine (JVM). The framework is designed to operate with a JVM that uses just-in-time (JIT) compilation to convert Java bytecode into native machine code.

Jazz2 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 to the user.

The core of Jazz2 is its support services and user-extensible test library of structural test strategies. The support services provide the common functionality that different structurals tests need, such as instrumentation, control flow analysis, memory allocation, and result collection. Jazz2 comes with a library of tests that we built using the framework. It also supports adding tests by extending existing ones or creating new implementations.

2.1 Support Services

Jazz2 offers support services to simplify building structural tests in the framework. The services include:

Instrumentation Jazz2 supports various types of instrumentation including permanent (inline) and transient instrumentation. Permanent instrumentation remains in the program for its entire execution, while transient instrumentation can be dynamically inserted and removed.

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, or when the JVM is about to exit. These callbacks serve as the interception points between the JVM and astructural 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. Jazz2 supports memory allocation in both the JVM's managed heap and operating system memory buffers (i.e., the native program heap and/or `mmap` space). These memory regions can be used for data and code.

Control Flow Analysis Structural tests often need to analyze control flow properties of a method to determine how best to instrument it. Jazz2 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.

2.2 Extensible Test Library

The power of Jazz2 is the ability to add new testing strategies 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 add it to the library.

The initial structural tests that we implemented and added to Jazz2's test library support testing at node (statement) and branch granularity. These tests demonstrate how Jazz2 provides support for two major implementation approaches for structural testing:

Static This approach adds permanent instrumentation inline in the generated machine code during JIT code generation.

Demand-driven This approach dynamically instruments a method to determine coverage. Transient instrumentation is inserted and removed along the path of execution.

Our current test library contains four combinations of these strategies: Static Node, Static Branch, Demand Node, and Demand Branch. It also includes three variations that use a static minimization algorithm developed by H. Agrawal to reduce the number of instrumentation points needed to gather coverage [12]. These tests are Static Node Agrawal, Static Branch Agrawal, and Demand Node Agrawal. We discuss the implementation of the tests in Section 4.

Creating a new test is straightforward. If it is a static or demand-driven test, the base class of an existing test in the library can be extended. Alternatively, a new test can be added by extending the superclass for structural testing.

2.3 Test Specification

Jazz2 is flexible because it offers the ability to specify where and how to test at the method level. The framework permits combining different structural tests 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 demand-driven instrumentation. This approach avoids the expense of multiple independent test runs.

To achieve this flexibility, Jazz2 provides a test specification language, called *testspec*. This language lets a user of the framework specify *rules* about how testing should be performed. The current version of testspec allows for specifying a class and method name to test, as well as the test types to apply to that method. In a dynamic programming environment like Java, the class and method names may not even be known until runtime. To avoid specifying every class and method name to test, testspec provides language support to describe name patterns and how to deal with unknown code.

Testspec has three wildcard operators to specify name patterns and how to handle unknown methods. To specify all methods in a class, we use a special `=all` flag. The asterisk globs class names. The $\star$ matches any class in a specific package. For instance, `java.util.*` will match `Vector` and `ArrayList` just as it

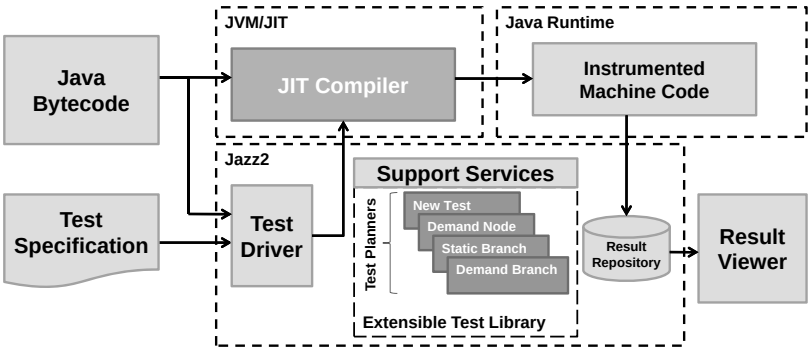


Figure 1. Jazz2 framework for structural testing.

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

Listing 1. Example test specification for *db* (SPECjvm98) where the six methods will be instrumented with demand branch testing and the remaining ones will be instrumented with static branch. Explicit rules have higher precedence than wildcard rules.

would in a Java import statement. However, when we need to test large projects or components, we may have a hierarchy of subpackages to test. Since the asterisk in Java does not import subpackages, we kept ours similarly limited. We use `**` to match 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 Listing 1. In the example, the first six lines explicitly state a (class name, method name) pair to be instrumented with the Demand Branch test from the Jazz2'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. Framework Implementation

Jazz2 is integrated with the Jikes RVM [13], a just-in-time compiler (JIT) for Java that is itself written in Java. A major challenge of a framework that is coupled to a separate code base (like Jazz2 and the RVM) is developing the framework with another code base that constantly and independently changes. The obvious temptation is to take the most recent release version and ignore any updates while developing your own component. This is not without disadvantages as any improvement or bug fix since the targeted version may be missed. However, the effort of porting the framework to new versions of the VM may not justify supporting every point release. Nevertheless, new major versions are often worth the effort. To ease the porting effort of Jazz2 to new versions of the RVM, it was designed to flexibly integrate with the RVM in as few places as possible.

Jazz2's test driver (see Figure 1) is implemented by the class `FrameworkHarness`. This class facilitates interaction with the RVM and isolates Jazz2 from it. Our previous experience in creating the original Jazz on top of an older version of the RVM led us to identify four areas in which Jazz2 needed to minimally interact with the existing code base or its output. First, inserting code as instrumentation means that we 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. Third, each test planner requires some facilities that may already exist in a compiler, such as control or data flow analysis. Finally, we need to efficiently parse and interpret our test specification language to specify what and where to test.

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 we 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 possible, it uses a bootloader with a minimally compiled bootimage to begin the RVM. Since the callbacks to `FrameworkHarness` are part of the bootimage, we must add `FrameworkHarness` to the bootimage. It is the only class in Jazz2 that appears in the bootimage. We take precaution to avoid instrumenting internal RVM code since the bootstrapping process is delicate and much of the Java class library is unavailable.

3.2 Instrumentation & Code Generation

We require access to the compiled method code to instrument it. To support this, there needs to be a low-level interface between the JIT compiler and Jazz2. There are four points during compilation where we might need to intercept control and have Jazz2 or one of its test implementations do work. These four events and the interface for capturing this interaction are shown in Table 1. We modified the RVM's JIT compiler to call these methods for a test implementation that registers a handler for the events.

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

The bytecode-oriented methods are used to insert static instrumentation directly into the generated machine code stream. The

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.

Table 1. An interface for adding instrumentation in a method-oriented JIT environment.

static test strategies in Jazz2's current test library implement the `onBytecode` event to insert probes at the beginning of a basic block. The `afterBytecode` event is provided to intercept the JVM after bytecode code generation. Our current tests do not use it.

The `afterCompilation` event is a natural place to put clean-up for static instrumentation. For demand-driven testing, it serves a more fundamental role. Demand-driven testing instruments the compiled machine code with *fast breakpoints* [19]. A fast breakpoint is a jump out-of-line to separate instrumentation code (i.e., an instrumentation probe and/or an instrumentation payload). A fast breakpoint can be dynamically added by overwriting existing code, and it can be easily removed by patching it with the original instruction it replaced. The `afterCompilation` event is used to intercept the JIT by an implementation of a demand-driven test in Jazz2. The demand-driven test implementation modifies the compiled instruction stream and returns the instrumented machine code to the RVM for execution.

3.3 Memory allocation

In general, we need four types of memory in Jazz2:

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 we extend the RVM to support Jazz2, the Java code in the test library that implements a test 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 Jazz2, 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.

We avoid 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), we simply choose to avoid the extra lookup costs and allocate space that is outside of the Java heap.

Most of the test strategies that we have implemented share some common code between instrumentation probes in order to reduce the 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. We reuse this functionality to allocate an executable page of memory to store the test payload code.

With demand-driven testing, we seek to overwrite as little of the original instruction stream as possible. If the instrumentation extends past the end of a basic block, the program might jump into the middle of that instrumentation and cause a crash. We do two things to make the instrumentation probes as small as possible. First, we extract the core functionality of the test probes into a single shared test payload. However, each probe needs to do slightly different work, and thus, the payload is essentially a parameterized function. We also need a place to set up the arguments and make the function call. Each probe has its own *trampoline*, which is a short code segment that sets up and makes a call to the payload. The instrumentation (probe) is a small jump overwriting the existing machine code that transfers control to a trampoline and eventually to the payload. In Jazz2, the trampolines are emitted directly to the end of the RVM's machine code object which contains the JIT-ed machine code. We use a relative jump for a fast breakpoint, allowing a test implementation to be unaffected by a copying GC.

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 may indicate to a demand-driven test where to dynamically insert new test probes or where to remove existing ones upon recording coverage information. The memory for a test plan should not move during execution, and so we can allocate it via `mmap`, or use `malloc` as the region does not need to be executable (it is data).

The final piece of memory needed for Jazz2 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 our 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.

To support memory storage in Jazz2, requires modifications to the RVM which have grown 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 we 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 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 [17]. 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.

3.4 Implementing Test Specification

A test specification, written in testspec, can be passed to Jazz2 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. Our 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 performance of the Java program under test. For example, *javac* in SPECjvm98 has 742 methods. A linear search for a matching test is responsible for 4% of total overhead.

We changed the intermediate representation 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. We implemented a solution where (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 we were able to reuse rather than writing our own.

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

3.5 JVM Support for Software Testing

In developing the support services and test library for Jazz2, we identified several places where a JVM developer can ease implementation of a tool like Jazz2.

A foundation of structural testing is to discover control flow properties of a region of code. To this end, we need, at bare minimum, a control flow graph (CFG). The RVM provides a CFG generator that scans the Java bytecode and determines the basic blocks and their predecessors. Jazz2 needs both predecessor and successor information. We extended the CFG with this information and other information that is useful about the structure of the code.

While implementing the structural testing optimizations suggested by Agrawal, we needed several analyses such as pre- and post-dominator information. 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 we wrote our own analyses from scratch. The implementation of Jazz2 would have been simpler with support for control and data flow analyses that is separated from the optimizer (e.g., such as Phoenix provides [8]).

Finally, a JVM could provide a rich and varied set of events to register callbacks. The RVM provides several callbacks, many of which we do not need, but the exit handler callbacks were useful for reporting our collected coverage results. For instance, the code generation interface of Section 3.2 would be convenient to achieve as much isolation from the JVM codebase as is possible.

4. Case Studies of Structural Testing with Jazz2

Jazz2 was built to support two significantly different strategies for implementing structural testing: static testing and demand-driven testing. To implement both strategies required the use of different events from the interface in Table 1. Figure 2 illustrates the interaction of the `FrameworkHarness` with the rest of the RVM for both static and demand-driven testing. Figure 2a shows static testing. `FrameworkHarness` takes the specification of what to test and interacts with the JIT during compilation. This produces a compiled method that contains instrumented code.

For demand-driven testing, shown in Figure 2b, the interaction happens differently. Demand-driven testing does not insert code into the compiled method but instead overwrites existing machine instructions with instrumentation. To that end, the interaction happens only at the end of compilation and the JIT-ed code serves as input to a demand-driven test planner.

```
1  public void onCompilation(
2        NormalMethod m) {
3
4     cfg = CFGBuilder.build(m);
5     seedSet = cfg;
6  }
7
8  public void onBytecode(
9        int bytecodeAddress,
10       Assembler asm) {
11
12    int index = seedSet.indexOf(
13       new Integer(bytecodeAddress));
14
15    if(index < 0) {
16       return;
17    }
18
19    asm.emitPUSH_Reg(GPR.EAX);
20    asm.emitMOV_Reg_Imm(GPR.EAX, plan[index]);
21    asm.emitMOV_RegInd_Imm(GPR.EAX, 1);
22    asm.emitPOP_Reg(GPR.EAX);
23 }
```

Listing 2. The static node coverage test planner.

4.1 Static Testing

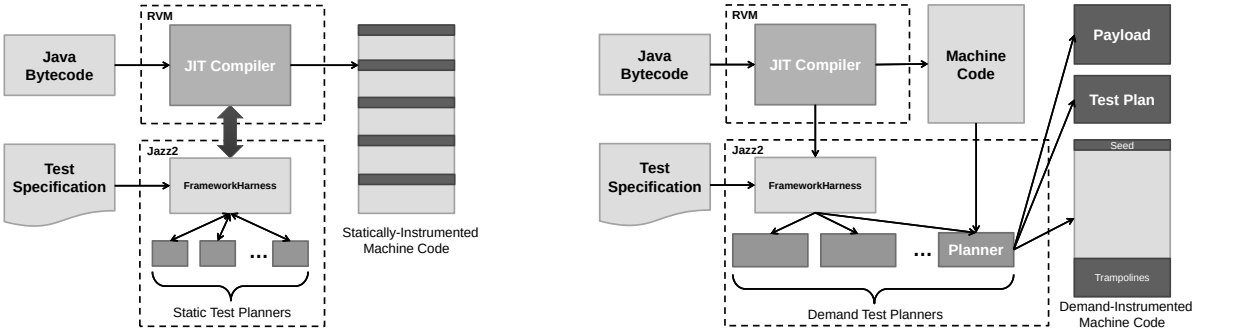
Figure 2a shows how static testing operates. A testspec specification is first loaded and parsed. When a method is about to be compiled, an `onCompilation` event is sent to the `FrameworkHarness`. If the current method matches a rule for a static test, Jazz2 instantiates a static test planner. The planner is responsible for instrumentation code generation, 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.

Listing 2 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.

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, Jazz2 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.



(a) Static instrumentation in Jazz2.

(b) Demand-driven instrumentation in Jazz2.

Figure 2. Jazz2 uses two different approaches for tying into the RVM depending on the style of instrumentation being done.

```
1 public void onCompilation(
2       NormalMethod m) {
3
4    cfg = CFGBuilder.build(m);
5    seedSet = Agrawal.getProbeSet(cfg);
6 }
```

Listing 3. Extending the basic static node coverage planner to incorporate Agrawal's probe reduction algorithm.

```
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
```

Listing 4. Static payload for branch coverage testing.

H. Agrawal developed an algorithm to reduce the number of instrumentation probes needed to record complete coverage [12]. The minimization algorithm requires control flow analysis on a method under test. We incorporated the algorithm as a support service since it can be used by more than one test strategy. With this service and the basic Static Node, we created a new test, Static Node Agrawal, that does node coverage with reduced instrumentation. We extended the basic Static Node planner to use the algorithm as is shown in Listing 3. 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 base class of the basic Static Node test were required.

We implemented a Static Branch coverage test in a similar fashion to Static Node. The `onCompilation` event is handled identically, as we still need to build a control flow graph. 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 Listing 2. Static Branch replaces this line with a call to the payload shown in Listing 4. Lines 1–2 load the identifier of the block that immediately preceded the current so we can mark that edge 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, we emit a call to the shared payload functionality.

Similar to Static Node, Agrawal's algorithm can be used to reduce the amount of instrumentation for branch coverage testing. We extended the basic Static Branch test planner to apply Agrawal's algorithm in the same as we did for Static Node.

4.2 Demand-Driven Testing

Demand-driven testing adds instrumentation after method compilation. Figure 2b shows the difference from static testing. `FrameworkHarness` implements the `afterCompilation` event handler. This handler checks whether the just compiled method matches a rule for a demand-driven test. If so, it invokes the `instrument` method of the appropriate demand-driven test planner.

Listing 5 shows the base class for demand-driven structural tests in Jazz2. Subclasses perform the appropriate work via the `instrument` method which is invoked from `FrameworkHarness`. The arguments to the `instrument` method 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-driven testing extends the three common elements from static testing and adds a fourth. All demand-driven tests have in common:

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

The Demand Node test planner is similar to the Static Node planner. It constructs a CFG for a method and determines a seed

```
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,
19          (byte) 0xE9);
20       instrs.set( nInsertAddr+1,
21          (byte)((nImm >> 0) & 0xFF));
22       instrs.set( nInsertAddr+2,
23          (byte)((nImm >> 8) & 0xFF));
24       instrs.set( nInsertAddr+3,
25          (byte)((nImm >> 16) & 0xFF));
26       instrs.set( nInsertAddr+4,
27          (byte)((nImm >> 24) & 0xFF));
28    }
29 }
```

Listing 5. The base class for all demand-driven structural tests.

set of locations to place initial instrumentation. 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 2b.

Demand-driven branch coverage is considerably more involved. As discussed in our previous work [21], there are certain control-flow structures that lead to problems with the dynamic nature of the demand-driven probes. If we do not fix the problem, there can be edges that are left with no instrumentation to record their coverage. We call this the *stranded block* problem.

The solution to the stranded block problem involves two special instrumentation types. The first causes probes to remain in the code associated with a stranded block until all edges involved in the stranded block are covered. The second type is used to record coverage for a block with a single incoming edge. Thus, in the Demand Branch test, there are actually three types of instrumentation payloads: a default payload, a self-recording payload, and a stranded block payload. These payloads require different parameters, and thus, three different kinds of trampoline.

5. Evaluation

The current Jazz2 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 collected 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.

We used SPECjvm98 [11] for benchmark programs. Our current implementation does not include testing strategies for multi-threaded programs, and as such, we discarded *mtrt*, which is a multi-threaded ray tracer. Details about the properties of the benchmarks are shown in Table 2. The table gives an indication of the complexity of each benchmark and the baseline runtime without structural testing. All timing results are averages across three runs of each benchmark.

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

Table 2. SPECjvm98 characateristcs. "Methods" is the number of executed methods. "Nodes" and "Edges" is the number nodes and edges in the CFG. "Runtime" is the baseline's execution time.

To evaluate Jazz2, we present three sets of results for seven coverage tests (Static Node, Static Node Agrawal, Demand Node, Demand Node Agrawal, Static Branch, Static Branch Agrawal and Demand Branch). We first experimentally examine the cost of structural testing, including the cost of test planning and execution of the instrumentation. Next, we study memory overhead. Finally, we examine how the tests impact garbage collection.

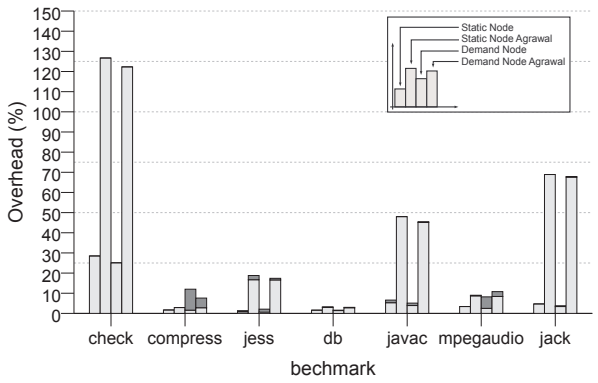
5.1 Performance Overhead

Because structural testing requires "extra work" when a program is executed, it imposes performance overhead. In the approach employed by Jazz2, 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.

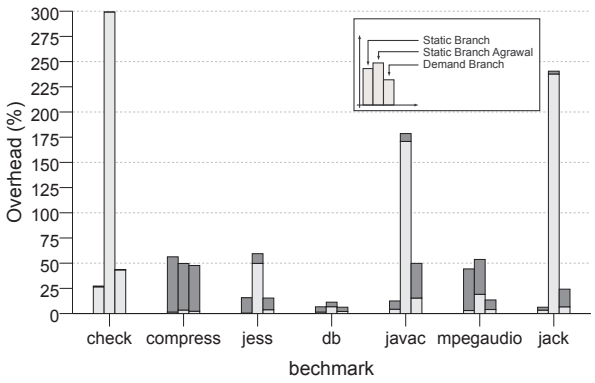
Figure 3 shows performance overhead of the SPECjvm98 programs for the seven test strategies. This overhead comes from test planning and the number of probes needed for a test. It also depends on the efficiency of Jazz2's instrumentation code. Figure 3a gives the overhead for node testing (Static Node, Static Node Agrawal, Demand Node, and Demand Node Agrawal). Figure 3b 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 2). The light gray part of each bar is the overhead from test planning and the dark gray part is the overhead from the instrumentation executed to gather coverage information.

The most apparent trend shown in the node coverage results of Figure 3a 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 testing approach does the most analysis of the seven strategies since it has to construct the CFG and compute dominator information to minimize the number of 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 Jazz2, this fixed cost is not well enough 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



(a) Overhead versus an uninstrumented run for node planning (shown in light gray) and instrumentation (shown in dark gray).



(b) Overhead versus an uninstrumented run for branch planning (shown in light gray) and instrumentation (shown in dark gray).

Figure 3. Overhead for Jazz2's node and branch coverage techniques broken down into planner cost and instrumentation cost.

instrumentation is only incurred once because the instrumentation is removed after only a few loop iterations.

Branch testing, shown in Figure 3b, has a similar trend in planning overhead as node coverage. The middle bar, representing Static Branch Agrawal, again shows the high cost of doing Agrawal's probe reduction technique. The Static Branch and Demand Branch (left and rightmost 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 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 performs worse than the static techniques. This result is due to two factors. The first 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.

It is interesting to note in the branch results that 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 Jazz2, 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 Jazz2 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.

5.2 Memory overhead

Jazz2 supports memory allocation from the JVM's managed heap and the operating system (i.e., outside of the JVM). Table 3 lists the memory requirements for the tests in Jazz2's current test library. The figures in the table are the sum of the payload code, trampoline code, test plan, and inline instrumentation.

The first four rows in the table detail the memory needs of the node coverage test 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 store a coverage result. The resulting total memory needs range 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. We find that 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, which gives 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 needing 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 to place the fast breakpoint at, 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

	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

Table 3. Total memory usage. This number includes the size of the payload, trampolines, test plan, and inline instrumentation.

should be placed. Despite the extra memory needs, our experiments only show a 2.1 times increase in memory usage over Static Branch.

The maximum memory used for code and instrumentation for any benchmark was under one third of a megabyte. From these results, we conclude that the library of tests that Jazz2 provides have reasonably small memory requirements to implement a test.

5.3 Impact on GC

Planning and performing a coverage test requires memory storage as shown earlier in Table 3. 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, Jazz2's memory is intertwined with the application memory, which can cause more pressure on garbage collection. This adverse interaction can be significant and harm program performance. In our 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 test probes.

To analyze Jazz2's influence on the number of garbage collections, we ran each of the seven testing strategies 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. For our baseline, we 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 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.

The remaining table rows show the net change in unforced garbage collections for each test strategy. Using Agrawal's technique causes many additional garbage collections due to 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 the demand-driven node testing techniques. In both cases, the instrumented version has *one less GC* than the baseline. This 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, causing 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 technique, the impact on the number of GCs was acceptably small. When we add in the Agrawal support service's work, we find a significant memory usage which additionally impacts runtime from doing the additional GCs. The previously proposed approach of doing offline planning or caching the results of planning for unchanged methods may be necessary to get reasonable performance out of the technique.

6. Related Work

Jazz2 is integrated with the Jikes Research Virtual Machine (RVM) [7, 13]. It uses the coverage testing techniques and lessons learned from our first version of Jazz [21]. Jazz [21] focused on demand-driven testing. It was not not an extensible framework for building and apply multiple test types. It was built with the older 2.0 series of Jikes RVM and the experience of moving to the 3.x versions directly influenced Jazz2's design.

There are several tools to collect coverage for Java programs. Commercial tools such as Clover [3] and IBM's Rational Suite [10] can collect node coverage and JCover [6], as well as open source tools such as Emma [5] and Cobertura [4], can collect both node and branch coverage information.

These tools take one of three strategies for collecting coverage information. One approach is to use the JVM interface meant for external debuggers. When this interface is used, JIT compilation is disabled and the bytecode is interpreted, unlike with Jazz2's ability to coexist with the JIT compiler. Alternatively, instrumentation is inserted into the class files with a tool such as the Bytecode Engineering Library (BCEL) [2] or at load time with a custom classloader. Neither of these approaches allows for the dynamic removal of the instrumentation as Jazz2 does.

Additionally, these tools apply instrumentation to entire class files or jar files. They lack the ability that Jazz2 has to choose the most appropriate instrumentation type on a per method basis.

Balcer, Hasling, and Ostrand propose a domain specific language to ease the overall testing process [14]. 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) [16] 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 regions of Java code for testing [22]. It is unclear if their language supports the application of different testing strategies to the selected regions. Jazz2's simple language supports this and the implementation provides it.

Fast breakpoints were pioneered by Kessler [19], but these breakpoints were proposed for traditional debugging rather than as dynamic instrumentation for structural testing. Dynamic instrumentation systems such as PIN [9], Dyninst [18] and Paradyn [20] used a

	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	—	—

Table 4. Change in the number of garbage collections. Top two rows (in italics) are number of collections for baseline without testing, including number forced calls to `System.gc()` and unforced collections. The other rows show the increase (+), decrease (-), or no change (—) in collections over the baseline. The default initial heap size of 20 MB and maximum heap size of 100 MB were used.

technique similar to ours to instrument programs. Like our framework, Dyninst is intended to be general, with a language for specifying instrumentation [18]. However, their instrumentation techniques were not designed to support test development.

Tikir and Hollingsworth use a dynamic technique for node coverage with Dyninst [18, 23]. As in Jazz2, the Dyninst tool dynamically inserts instrumentation on method invocations for node coverage. However, instead of removing instrumentation as soon as possible, a separate thread periodically removes the instrumentation. This instrumentation remains until collected, even when it is not needed. Additionally, Tikir and Hollingsworth do not address the issues of test specification nor is their tool a framework for implementing additional tests such as branch coverage. Jazz2's extensibility permits this technique to be implemented with the support services. We have an implementation of collection-based removal of test probes currently in development.

Jazz2 makes use of the work of Agrawal to reduce the number of instrumentation probes inserted at compile-time [12]. Other strategies exist for reducing the cost of collecting coverage information, such as path profiling [15]. This technique inserts probes that add a block-specific value to a counter so that an accumulated sum uniquely identifies a taken path. Path coverage subsumes both node and branch coverage, but the probes inserted are never removed. This technique could be added to Jazz2's test library.

7. Conclusion

This paper described Jazz2, a flexible and extensible framework for structural testing. Jazz2 provides support services to do code generation, memory management, control flow analysis, and handle interaction with a JVM. Using these services we built an initial library of test planners to do structural testing. New test planners can be created by extending existing tests from the library or writing new ones with the support services. We detailed our experience in Jazz2 on top of IBM's Jikes RVM for Java. Our evaluation shows that Jazz2 has feasibly low overhead. It is a powerful framework that provides a rich set of capabilities to create a variety of structural tests relying on different instrumentation strategies.

References

[1] The AspectJ project. `http://www.eclipse.org/aspectj/`.

[2] Bytecode Engineering Library. `http://jakarta.apache.org/bcel/`.

[3] Clover. `http://www.cenqua.com/clover/`.

[4] Cobertura. `http://cobertura.sourceforge.net/`.

[5] EMMA. `http://emma.sourceforge.net/`.

[6] JCover. `http://www.codework.com/JCover/`.

[7] Jikes RVM. `http://jikesrvm.org/`.

[8] Microsoft Phoenix. `http://research.microsoft.com/phoenix`.

[9] Pin. `http://www.pintool.org/`.

[10] IBM Rational PurifyPlus. `http://www.ibm.com/rational`.

[11] Standard Performance Evaluation Corporation. `http://www.spec.org/jvm98`.

[12] H. Agrawal. Dominators, super blocks, and program coverage. In *Symp. on Principles of programming languages*, pages 25–34, 1994.

[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] 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.

[15] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, 1994.

[16] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[17] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev. Demystifying magic: high-level low-level programming. In *Int'l. Conf. on Virtual Execution Environments*, VEE '09, pages 81–90, 2009.

[18] J. K. Hollingsworth, O. Niam, B. P. Miller, Z. Xu, M. J. R. Goncalves, and L. Zheng. Mdl: A language and compiler for dynamic program instrumentation. In *Int'l. Conf. on Parallel Architectures and Compilation Techniques*, PACT '97, pages 201–, 1997.

[19] P. B. Kessler. Fast breakpoints: design and implementation. In *Conf. on Programming language design and implementation*, pages 78–84, 1990.

[20] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *Computer*, 28:37–46, November 1995.

[21] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa. Demand-driven structural testing with dynamic instrumentation. In *Int'l Conf. on Software engineering*, pages 156–165, 2005.

[22] H. Rajan and K. Sullivan. Aspect language features for concern coverage profiling. In *Int'l. Conf. on Aspect-oriented software development*, AOSD '05, pages 181–191, 2005.

[23] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *Int'l. Symp. on Software testing and analysis*, pages 86–96, 2002.

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