Systematic Testing of Multithreaded Java Programs

Derek L. Bruening

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 21, 1999

Author ______________________________

Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by ______________________________

John Chapin
Thesis Supervisor

Accepted by ______________________________

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Systematic Testing of Multithreaded Java Programs

Derek L. Bruening

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 21, 1999

Abstract

Concurrent programs are becoming common, while testing techniques that can adequately test such programs are not widely available. Due to the nondeterministic nature of concurrent programs, program errors resulting from unintended timing dependencies can be extremely difficult to track down. We have designed, proved correct, and implemented a testing algorithm called ExitBlock that systematically and deterministically finds such errors. ExitBlock executes a program or a portion of a program on a given input multiple times, enumerating meaningful schedules in order to cover all program behaviors. Other systematic testers exist for programs whose concurrent elements are processes that run in separate memory spaces and explicitly declare what memory they will be sharing. ExitBlock extends previous approaches to multithreaded programs in which all of memory is potentially shared. A key challenge is to minimize the number of schedules executed while still guaranteeing to cover all behaviors. Our approach relies on the fact that for a program following a mutual–exclusion locking discipline, enumerating possible orders of the synchronized regions of the program covers all possible behaviors of the program. This thesis presents ExitBlock and its correctness proof, describes its implementation in the Rivet virtual machine, and demonstrates its ability to detect errors in actual programs.

Thesis Supervisor: John Chapin

Title: Assistant Professor of Electrical Engineering and Computer Science

Acknowledgments

This research was supported in part by the Defense Advanced Research Projects Agency under Grant F30602-97-2-0288. Additional support was provided by an equipment grant from Intel Corporation.

The Rivet Virtual Machine is a project of the Software Design Group at the Laboratory for Computer Science at MIT. Like all large projects it is the result of a team effort. Contributors to the project (besides myself) have included John Chapin, Brian Purville, Ian Schecter, Gong Ke Shen, Ben Walter, and Jon Whitney. It has been a pleasure working with all of them. Other than Rivet's thread scheduler, extra fields, and checkpointing, all of the work on Rivet discussed in this thesis is joint work with other members of the Rivet team.

My implementation of the Eraser algorithm was modeled on Delphine Le Cocq's implementation for Kaffe.

I would like to thank Daniel Jackson and Martin Rinard for taking the time to answer my questions regarding related work in the field.

I would like to thank my thesis advisor, John Chapin, for all of his advice, encouragement, and constructive criticism. He has always been available and willing to discuss anything, and his enthusiasm is refreshing.

A warm thank–you to Barbara Cutler for making me start writing this thesis early, for her help in making my figures readable, and for her help in proofreading.

Contents

Chapter 1

Introduction

Concurrency is often necessary in implementing systems that receive and react to multiple simultaneous requests. Nearly all web servers, database servers, and operating systems exhibit concurrency. In addition, applications that utilize graphical interfaces (web browsers, word processors, Java applets) often use concurrency. Concurrency is increasingly used for applications with high reliability requirements (such as air–traffic control software).

Concurrent programs are more difficult to test than sequential programs. Traditional testing techniques for sequential programs consist of test suites that simply run the target program on different sets of representative inputs. Such tests are likely to cover only a small fraction of the possible execution paths of a concurrent program. This is due to the nondeterministic nature of concurrent programs: merely controlling the program's input is not enough to control the program's behavior. We define a particular program execution's *behavior* to be the set of externally visible actions that the program performs during its execution.

A concurrent program consists of components that execute in parallel. These components are of two varieties: processes, which run in separate memory spaces, and threads, which all share the same memory space. Different orders of the components' actions with respect to one another may result in different program behaviors; the resulting erroneous behaviors are called *timing–dependent errors*. The order of components' actions is often nondeterministic due to asynchronous input or interrupts or simply a nondeterministic *scheduler*. The scheduler is the operating system or library component that determines at run time the actual order of execution, or *schedule*, of the processes or threads. Even if the schedule is deterministic for a particular scheduler executing the program, it is possible and indeed likely that a different scheduler's execution of the program will result in a different schedule.

The potential presence of timing–dependent errors and nondeterministic scheduling means that the fact that a concurrent program passes a traditional test suite says little about whether it will always function properly wherever executed. Even when a timing–dependent error is found, there is no guarantee that the erroneous behavior can be repeated by re–executing the program with the same input, making debugging concurrent programs also very difficult.

In this thesis we describe a practical algorithm for enumerating the possible behaviors of a section of a multithreaded Java program, for a given input. By systematically exploring a concurrent program's behaviors, all erroneous behaviors can be found; furthermore, a particular schedule leading to an error can be recorded and replayed. We have built a systematic tester that implements this algorithm. Our tester is implemented in Java on top of the Rivet Virtual Machine, a Java virtual machine built by our group. Currently the tester can execute over 80 schedules per second and can test small execution regions (consisting of two to four threads with perhaps thirty or fewer synchronized regions each) in a matter of minutes. This needs to improve in order to be able to test larger programs; we discuss how to achieve this in Chapter 5. We show that the tester finds common multithreading errors that are often difficult to detect; for example, errors that occur only in certain orderings of modules that are individually correctly synchronized, or using an `if` instead of a `while` to test a condition variable.

This thesis is organized as follows. The rest of this chapter details the specific objectives of the systematic tester and describes related work. Chapter 2 describes the algorithms used by the systematic tester and proves their correctness. Chapter 3 describes the Rivet Virtual Machine in brief, and the implementation of the tester. Chapter 4 presents the results of running the tester on a number of sample programs. Chapter 5 summarizes the work and discusses future improvements. Finally, Appendix A discusses the implementation of Rivet in detail.

1.1 Objectives

Our goal was to build a tool that could systematically enumerate all possible behaviors of a section of a multithreaded Java program, for a given input. The ubiquity of Java programs utilizing graphics and user interfaces combined with Java's easy–to–use threading and synchronization mean that quite a few Java programs are multithreaded, making such a tool very useful.

One way to find all behaviors is to force the program through all possible schedules of its processes or threads (in our case, threads). The problem with this is that the number of schedules increases exponentially with program size so quickly that testing even very small program sections is intractable. However, not all schedules produce different behaviors. By identifying behavior–equivalence classes of schedules and executing as few members of the same class as possible (only one member of each class is necessary to cover all program behaviors) we can reduce the number of schedules enough that we can test usefully sized program sections.

Our tester uses this idea to systematically enumerate the behaviors of a multithreaded Java program that meets certain requirements on a given input. The requirements are detailed in Section 2.1; in brief, they are that the program follows a mutual–exclusion locking discipline, that its finalization methods are "well-behaved," and that all of its threads are terminating. The tester requires no separate model or specification of the program and does not require the source code to be available — it does its testing dynamically by running the actual program.

The completeness of the enumeration is defined in terms of *assertion checking*. The guarantee is that if there is some execution of a program on a given input that leads to an assertion failure, the tester will find that execution. Since any observable program behavior can be differentiated from all other behaviors by assertions, the tester effectively guarantees to execute all behaviors of a program. This allows a user to detect all erroneous behaviors. In addition, the tester detects deadlocks; it provides a guarantee to find a deadlock if one exists.

The exponential increase in the number of schedules with program size means that the tester cannot test entire programs in a reasonable amount of time (a few hours). Combining the tester with a debugger, however, will allow a user to systematically test a program from an execution point that the user believes is just prior to the part of the program that needs to be tested. Our group is building a debugger that will be integrated with the tester.

The tester will generate tracing information that can be used to replay specific schedules. The tester could be run overnight and in the morning all schedules leading to assertion failures or deadlocks could be replayed and examined.

1.1.1 Example Target Program

As an example of a simple program exhibiting one sort of concurrency error that the systematic tester could help find, consider the sample Java program `SplitSync` shown in Figure 1.1. `SplitSync` creates two threads that each wish to increment the field `x` of the variable `resource`. The Java `synchronized` statement limits its body to execute only when the executing thread can obtain the lock specified. Thus only one of the two threads can be inside either of the synchronized statements in the program at one time.

The way in which the threads execute the increment of `resource.x` is very roundabout; however, it is representative of a common class of errors in large systems in which the processing of a resource is split between two modules. The intention is that no other module can access the resource while these two modules (call them A and B) are processing it. Module A acquires the resource (via a lock) and performs the first stage of processing that resource. Wishing to let module B finish processing the resource, A releases the lock. Module B immediately grabs the lock and continues processing the resource. However, some other module could acquire the resource and modify it after A releases the lock but before B obtains it, violating the invariant that the resource is never seen in a partially processed state by anyone but A and B.

In a similar manner, `SplitSync` incorrectly synchronizes access to the shared variable `resource` by allowing one thread's increment of `resource.x` to be interrupted by the other thread. This is a race condition: the two threads could both set `y` to the old value of `resource.x`, and then both set `resource.x` to be `y+1`, resulting in one of the increments being lost and an incorrect final value for `resource.x`.

A traditional test suite might never find this bug because the thread scheduler of its Java virtual machine may never perform a thread switch while either thread is between the two synchronized blocks. Even if the bug is found, it may be very difficult to duplicate. A user of the code could have a different scheduler that switches threads more often, and so

```java
public class SplitSync implements Runnable {
  static class Resource { public int x; }
  static Resource resource = new Resource();

  public static void main(String[] args) {
    new SplitSync();
    new SplitSync();
  }

  public SplitSync() {
    new Thread(this).start();
  }

  /** increments resource.x */
  public void run() {
    int y;
    synchronized (resource) { // A
      y = resource.x;
    }
    synchronized (resource) { // B
      // invariant: (resource.x == y)
      resource.x = y + 1;
    }
  }
}
```

Figure 1.1: Sample Java program `SplitSync` illustrating an error that our tester can help find. Although all variable accesses in this program are correctly protected by locks, it contains a timing–dependent error. By splitting the increment of `resource.x` into two `synchronized` statements, an error will occur if the two threads are interleaved between the `synchronized` statements. Both threads will read the original value of `resource.x`, and both will then set it to one plus its original value, resulting in the loss of one of the increments.

the behavior the user observes could be quite different from the results of a traditional test suite. The systematic tester would be able to find the bug by systematically enumerating schedules of the program.

This example shows that it can be difficult to test even small concurrent programs with traditional test suites. We now discuss other methods of testing concurrent programs.

1.2 Related Work

There is a large body of work in the general area of testing concurrent programs. It can be divided into categories based on five criteria. Table 1.1 on page 24 classifies the related work discussed in this section into these criteria.

The first criterion is the sorts of errors that are detected. In testing a program, one would like to ensure that for every input the correct, or desired, output is always generated and that no incorrect output is ever generated. There are several ways to do this; the most common is to compare the actual output observed with the desired output. Because we may be interested in testing a section of a program that contains no output, let us generalize and say that in testing a program we would like to state some assertions or invariants about the program and have the tester detect violations of those assertions or invariants. Since assertions about output can be made, detecting assertion violations can detect every error that comparing outputs can detect. In addition to detecting program–specific errors, detecting common concurrency errors can be very useful. Deadlocks, data races, and starvation are all commonly looked–for conditions. Focusing on a single class of errors can achieve greater efficiency and better guarantees on finding all such errors with no false alarms. Our systematic tester checks for assertion violations and detects deadlocks.

A second criterion is whether a testing method checks that a program has some property for all inputs or merely tests a program on one input. With the latter, test cases need to be generated that cover all inputs on which the program should be tested. Our tester tests a program for a single input. (See Section 1.2.7 for a discussion of test case generation for use with our tester.)

A third criterion is what guarantee a tester provides that the errors it finds are all of the errors present in the program. A tester that certifies its results are complete and considers all possible inputs (the second criterion) is called a *verifier*. The systematic tester guarantees its results but considers only one input, so it is not a verifier.

A fourth criterion is whether the tester works on the actual program or requires a specification or model of the program. Our tester works on the program itself.

Fifth and finally, we can classify testers based on whether they perform static analysis or actually run the program to be tested. The systematic tester runs the program.

We now survey the field of concurrent program testing.

1.2.1 Model checking

Model checking tools, or model checkers, verify properties of models of concurrent systems. These models are expressed in special–purpose modeling languages. Model checkers typically enumerate all reachable program states through the model and check properties of those states. They are verifiers — that is, they certify that the model has certain properties for all inputs. Inputs are typically not handled separately, but are considered implicitly by examining all state transitions of the model.

Model checkers view the task of finding the reachable program states as *state–space exploration*. The state space of a program is the set of all program states. Model checkers find the possible paths through the *state graph* of a program; the graph's nodes are program states, so following possible transitions between states leads to the set of reachable states. The fundamental challenge of exploring the state space of a concurrent program is the excessive number of paths. For a deterministic sequential program with no concurrency there is only one path for any given input. In contrast, concurrent programs' potential schedules lead to an exponential explosion in the number of possible paths. Even for concurrent programs of moderate size, testing every single path is usually infeasible. Thus model checkers must trim the search space by identifying paths through the graph that are equivalent in terms of their effect on program behavior, and follow as few paths as possible from each set of equivalent paths.

State–space caching algorithms [GHP95] are one method for pruning the number of paths that must be searched. By remembering previously visited states, a model checker can identify paths that overlap and avoid searching the same path segment more than once.

Model checkers utilizing state–space exploration have been shown to be effective in verifying many aspects of models of concurrent systems, including checking for violations of user–provided assertions or invariants and for finding problems such as deadlocks. Unfortunately, they are restricted to verifying properties of a model's abstraction of the system. These verifications provide no proof that an implementation of the system satisfies those properties.

Our objectives were to build a tester that operates on an actual program and does not require a model of the program to be built. This means that model–checking technology cannot be directly applied to our problem. State–space exploration of an actual implementation of a system is more challenging than exploration of a model's state space. A model specifies exactly what information defines a program's state; the brevity of this information leads to a manageable number of states, which can each be given a unique identifier. These identifiers can be used in the detection of previously seen states for the purpose of path pruning (e.g., state–space caching). In contrast, an arbitrary program's state is defined by the values of all variables that could influence the behavior of any of that program's components. This is usually more information than can be encoded in a way that allows rapid identification of previously–visited states. Thus a tester that attempts to explore the state space of a program cannot make use of caching or other methods of pruning that require state identification.

1.2.2 Static Analysis

Static analysis of a program involves deducing properties of the program by examining its source or object code. These approaches are limited in what aspects of the program they can verify in reasonable amounts of space and time. Attempts to emulate the behavior of the program on all possible inputs typically end up consuming far more resources than actually running the code on real inputs.

Many static analysis tasks have been shown to be intractable [Tay83]. Indeed, static analysis techniques for detecting concurrency errors must act on an approximation of the reachable program states. In syntax–based static analysis, a flow graph is constructed for each concurrent component of the program. This flow graph is very simplified, containing only synchronization and branch behavior. Using these flow graphs, a master graph of the concurrency states of the program can be built. These states contain synchronization information but no data values. Various concurrency errors such as deadlocks can be detected by examining this master graph. However, the approximations made often result in the master graph containing states that are actually unreachable (by ignoring data values, branch predicates cannot be predicted), leading to spurious error detection.

Another type of static analysis is *symbolic execution.* Symbolic execution builds flow graphs that preserve as much data value information as possible, in order to compute branch predicates. Data races can be detected in the resulting master graph. However, because it has to consider all possible data values, the exponential explosion is tremendous, greatly limiting the applicability of symbolic execution.

Attempts have been made to combine syntax–based static analysis with symbolic execution [YT88]. The syntax–based static analysis directs the symbolic execution to try only the (relatively) few paths that the static analysis produces, while symbolic execution prunes the unreachable states from the static analysis results. This has had some successes. However, all static analysis suffers from the intractability of accurately analyzing program execution paths statically. These methods work only on certain types of programs. The presence of pointers or heap references in a program (especially in a language that is not strongly–typed) causes great loss of accuracy since static analysis cannot always identify the identity of objects if they are subscripted by variable expressions or referred to through a chain of references.

Thus even though static analysis can be successful in detecting deadlocks and data races, it cannot detect arbitrary assertion violations. Our tester is purely dynamic. We have found, however, that we could benefit from some static analysis information. This is discussed in Section 2.6.

1.2.3 Nondeterministic Testing

Nondeterministic testing of a program involves selecting inputs as test cases and for each input simply running the program multiple times. The idea of this repeated testing is that by running the program repeatedly, more of the possible schedules of the processes or threads that make up the concurrent program will be executed, increasing the chance that timing–dependent errors will be found. Random delay statements can also be inserted into the code to vary the timing of each run, which further increases the chance that timing–related conditions will be reached.

Nondeterministic testing may (and is likely to) end up executing the same schedule many times. Also, there is absolutely no guarantee that all behaviors will be covered. One method of attempting to solve this problem is to allow the user control over the scheduler, letting him or her vary the scheduling of each run. Because there are far too many schedules to try them all, the user must select those that he or she believes will lead to different behavior. Like the random delay statements, since this is not systematic there are no guarantees of complete coverage.

Generating test cases for nondeterministic testing of concurrent programs is very different from generating test cases for sequential programs. Inputs should be selected based not only on how representative they are of real inputs and their effect on program output but also on their influence on the scheduling of the program's concurrent components. Much work has been done in this field, both in generation of test cases [KFU97] and in evaluating coverage and quality of test cases [F+96]. Many test case generators require separate specifications of the program to be tested in order to work, since they need more information than can be gleaned statically from the program itself. Our systematic tester greatly simplifies the generation of test cases. This is discussed in Section 1.2.7.

Even when nondeterministic testing identifies an erroneous program behavior, finding the bug can be very difficult. For one thing, standard debuggers can only debug one process, so separate, non–communicating debuggers must be used on a multi–process program. Also, duplicating the error itself can be a challenge. For a sequential program, simply re–executing the program under a debugger will duplicate the error and allow it to be examined. However, re–executing a concurrent program may not duplicate the error due to nondeterminism. As mentioned earlier, our systematic tester can record and replay schedules that it executes. The ability to re–execute a program in exactly the same manner in which it was first executed is called *deterministic replay*. The Rivet Virtual Machine provides the tester with this ability (see Section A.7). Related work includes a modified Java virtual machine that provides deterministic replay of multithreaded Java programs [CH98], an incremental checkpoint/replay system [NW94], and *event history debugging* [MH89]. Event history debugging records the event history of a program at run time and allows for browsing and replay of the program's schedule. However, if a user found an error while executing outside any of these tools, he or she would have to repeatedly execute the program in the tool's environment until the error showed up again. The user needs to execute our tester only once to find the schedule that led to the error.

Nondeterministic testing also includes other testing methods in which each run cannot be repeated deterministically. There are tools that detect classes of concurrency errors but are nondeterministic. One example is Eraser [S+97], a dynamic data race detector. Eraser assumes that programs follow a mutual–exclusion locking discipline wherein each shared variable is protected by a set of locks that is held consistently when that variable is accessed. The tool runs the program on some input and detects cases where a variable is accessed with an inconsistent lock set (see Section 2.1.1 for a full description of the Eraser algorithm). However, the tool cannot guarantee that no data races are present since the program execution is nondeterministic (see Figure 2.2 on page 29 for an example of a program whose data race is nondeterministically detected by Eraser).

1.2.4 Specialized Domain Testing

When a limited or specialized concurrent language is being tested, specialized techniques can be employed. For example, there is a tool called the *Nondeterminator* [FL97] that detects data races in Cilk, a multithreaded variant of C. The Nondeterminator guarantees that if a program has a data race it will find and localize it. The Nondeterminator's algorithms depend on being able to model the concurrency of a Cilk program with a directed acyclic graph. Cilk threads cannot communicate with their peers, only with their parents. The methods used do not apply to the more general concurrency found in Java.

1.2.5 Deterministic Testing

In deterministic testing [CT91], test cases consist not just of inputs for the program to be tested but of (input, schedule) pairs, where each schedule dictates an execution path of the program. For each test case, the tester forces deterministic execution of the program using the schedule provided for that case and supplies the provided input to the program. This deterministic execution is repeatable; techniques such as those mentioned in section 1.2.3 are used to provide deterministic replay.

The problem with deterministic testing is in generating the test case pairs. Typically static syntax–based analysis is used, which is limited in the scope of what it can determine about a program. Approximations must be made and the resulting test cases are not guaranteed to cover all behaviors of the program. Using symbolic execution in determining test cases can help to some extent, but symbolic execution ends up duplicating work done during execution and using far more resources doing it. Given this, why not just execute the program in the first place and have some sort of feedback loop for dynamically generating new test cases? This turns deterministic testing into a combination of nondeterministic testing (just run the program up to a certain point) and deterministic testing (from that point on, record every schedule that should be executed to cover all program behaviors). This combination is called reachability testing [HTH95]. It can be made to cover all behaviors of a program for a given input. Reachability testing will be discussed further in the next section.

1.2.6 Behavior–Complete Testing

Behavior–complete testing of a program on a given input involves systematically executing every possible behavior of the program on that input. The problem with simply enumerating the possible schedules of the program is the exponential increase with program size in the number of schedules that must be considered. The solution is to figure out which schedules lead to the same behavior and to execute as few schedules for each behavior as possible. There is usually still an exponential increase in the number of schedules considered, but it can be reduced enough to test moderately sized programs.

Verisoft [God97] takes the ideas of model checking and applies them to checking actual programs that do not have models. It makes use of state–space pruning methods that do not require manageable state identifiers [God96]; for example, it does not reorder two sections of code that do not access any variables in common. It performs some static analysis to provide information needed by these pruning methods. Verisoft's target programs are small C programs consisting of multiple processes. Verisoft uses "visible" operations (operations on shared objects) in the code to define the global states of the program. Since processes run in separate memory spaces and must explicitly declare the variables that they share, the number of global states is significantly smaller than the number of states, and since processes can only communicate through global states, Verisoft need only consider schedules that differ in their sequences of global states. Verisoft further prunes the state space by identifying schedules that lead to identical behaviors.

Verisoft allows annotations to be inserted into the C code it is testing, including assertions that will be verified during testing. Verisoft has been shown to successfully discover errors in C programs composed of several concurrent processes. However, its techniques cannot readily be transferred to Java. Since Java contains threads which share the same memory space, all variables are potentially shared. Thus there are not a small number of explicitly declared "visible" operations that can be used to dramatically reduce the state space. Verisoft would have to assume that all variables are shared, and end up searching a prohibitively large state space.

Another behavior–complete testing method is the reachability testing mentioned in the last section. This builds on a method for systematically executing all possible orderings of operations on shared variables in a program consisting of multiple processes [Hwa93]. It is similar to Verisoft but does not do any pruning. It does parallelize the executions of different schedules to speed up testing. Reachability testing, like Verisoft, requires explicit declarations of which variables are shared and hence does does not apply directly to multithreaded programs.

1.2.7 Summary of Related Work

Table 1.1 summarizes the categories of methods for testing concurrent programs discussed in this section. It lists how they fit into each of the five criteria presented in the introduction to this section.

Only behavior–complete testing satisfies our criteria stated in the introduction to Section 1.2. However, existing behavior–complete testers do not operate on programs with multiple threads. No existing methods can solve our problem for us – none can guarantee to detect all assertion violations in a multithreaded Java program, for a given input.

We can make use of some of the ideas of these testers and the other methods. By combining Eraser with behavior–complete testing we can cut down on the exponential explosion in the number of schedules we need to consider. If we assume that a program follows a mutual–exclusion locking discipline, we can make use of the fact that enumerating possible orders of the synchronized regions of the program covers all behaviors of the program. We can even run Eraser in parallel with the tester to ensure that the program follows the discipline.

We can use some of Verisoft's techniques to further reduce the number of schedules we consider. For example, our algorithm does not reorder two sections of code that do not access any variables in common. As discussed in Section 2.6, in the future we plan to make use of static analysis to give the tester further information with which to reduce the search space. Also, we could parallelize the tester as reachability testing suggests.

The tester will provide deterministic replay of any schedule, in order for a user to

Method	Detects	Inputs	Complete?	Requires Model?	Static or Dynamic?
Model checking	Assertion violations, others	All	Yes	Yes	Static
Static Analysis	Data races, deadlocks, others	All	No	No	Static
Nondeterministic Testing	Assertion violations	One	No	No	Dynamic
Eraser	Data races	One	No	No	Dynamic
Nondeterminator	Data races	One	Yes	No	Dynamic
Deterministic Testing	Assertion violations	One	No	No	Both
Behavior–Complete Testing	Assertion violations	One	Yes	No	Dynamic

Table 1.1: Summary of properties of various methods for testing concurrent programs that are described in this chapter. A *complete* method guarantees to find all errors (of the type it detects) and not to find any false errors.

examine in more detail (perhaps with a debugger) paths that led to errors. The techniques of deterministic replay used in other systems are applicable. Section A.7 describes the Rivet Virtual Machine's approach to deterministic replay.

Since the tester tests a program for a single input, test case generation is crucial. Generating test cases for use with the tester is simpler than for other multithreaded program testing methods since the effects of different inputs on scheduling can be ignored. Conventional sequential program test case generation can be used instead of the complex generation techniques needed to generate test cases for nondeterministic and deterministic testing.

Chapter 2

Systematic Testing Algorithms

This chapter describes the algorithms used by our systematic tester, which enumerates all behaviors of a target multithreaded Java program on a single input. When the target program meets certain requirements, the tester guarantees to find all possible assertion violations in the program. Any program condition can be detected using assertions; thus, the tester guarantees to enumerate all possible behaviors of the program.

This chapter is organized as follows. Section 2.1 explains the testing criteria that the tester requires of client programs. Section 2.2 motivates and Section 2.3 presents and proves correct the ExitBlock algorithm which systematically explores all behaviors of a program. Section 2.4 modifies ExitBlock to create a more efficient algorithm, ExitBlock-RW. Section 2.5 shows how to detect deadlocks using ExitBlock or ExitBlock-RW in the ExitBlock-DD and ExitBlock-RWDD algorithms; it proves that ExitBlock-DD will detect a deadlock in a program if one is present. Finally, Section 2.6 discusses enhancements that could be made to these algorithms.

2.1 Testing Criteria

The tester requires that a program it is testing meets three criteria: that it follows a mutual–exclusion locking discipline, that its finalization methods are "well-behaved," and that all of its threads are terminating. Our implementation of the tester also assumes that the program's native methods meet the requirements of the Rivet Virtual Machine (see Section 3.1.2) and the tester's deterministic replay (see Section 3.3.2), but these requirements are implementation–dependent. Collectively we refer to the three fundamental requirements as the *testing criteria*. They are discussed in turn in the following three sections.

2.1.1 Mutual–Exclusion Locking Discipline

A mutual–exclusion locking discipline dictates that each shared variable is associated with at least one mutual–exclusion lock, and that the lock or locks are always held whenever any thread accesses that variable. Java encourages the use of this discipline through its synchronization facilities. The Eraser algorithm by Savage et al. [S+97] (described below) can be used to efficiently verify that a program follows this discipline.

This criterion can be verified by the tester, by running Eraser in parallel with itself to ensure that the discipline is followed. Running Eraser alone only ensures that a program follows the discipline in one particular schedule (see Figure 2.2), while running Eraser in parallel with the tester checks that the discipline is followed in every schedule. Even though the tester does not guarantee to execute all behaviors of a program that contains a discipline violation, it does guarantee to find the violation. This is guaranteed because the tester will not miss any behavior until the instant one thread communicates with another thread using shared variables that are not protected by consistent sets of locks. At this point Eraser will catch the discipline violation.

While errors of the sort detected by Eraser are frequent, they by no means exhaust the possible errors in a multithreaded program. Consider the example program `SplitSync` from Figure 1.1 on page 15, or any of the programs in Chapter 4. These are simple programs that correctly follow a mutual–exclusion locking discipline yet contain serious thread–related errors.

By limiting the tester to those programs that follow this locking discipline, some classes of valid programs are ruled out for use with the tester. For example, Java's `wait` and `notify` facilities can be used to build a *barrier*, which is a point in the program which all threads must reach before any threads are allowed to continue beyond that point. A program using barriers can validly use different sets of locks for protecting the same variable on different sides of a barrier. However, as Savage et al. argued, even experienced programmers with advanced concurrency control mechanisms available tend to use a simple mutual–exclusion locking discipline. We expect that requiring such a locking discipline will not overly restrict the applicability of the tester.

This assumption is the key to our testing algorithm. The reason we require it is explained in Section 2.2. We now briefly describe the Eraser algorithm.

The Eraser Algorithm

Violations of a mutual–exclusion locking discipline are called *data races*. The Eraser algorithm detects data races by dynamically observing what locks are held by each thread on each variable access. The discipline is followed only when a non–empty set of locks is associated with each shared variable, and that set (or a superset of it) is consistently held by every thread that accesses that variable.

The algorithm is implemented by instrumenting every lock acquisition and release and every variable read and write. The set of locks that each thread currently holds is tracked. A data structure for each variable v holds what is referred to in [S+97] as *C(v)*, the set of candidate locks for v. A lock is in *C(v)* at the current point in the program if every thread up to that point that has accessed v held the lock when it did so. *C(v)* is initialized to be the set of all possible locks, and on each access of v, *C(v)* is assigned to be the intersection of the old *C(v)* with the set of locks held by the thread accessing v. This is called *lockset refinement*. If *C(v)* ever becomes empty Eraser reports a violation of the locking discipline.

Eraser relaxes the locking discipline a little by allowing variables to be written while holding no locks in a constructor of the object that owns the variable, or in other initialization code, if the program has no concurrent access to objects until they are fully initialized. Eraser also allows access to read–only variables while holding no locks. To handle these cases where a variable need not be protected by locks, Eraser keeps track of a state for each variable. The states and their transitions are shown in Figure 2.1. As the figure indicates, Eraser assumes that the first thread to access a variable is initializing the variable, and that no locks are needed for this initialization. Only when a second thread accesses the variable does lockset refinement begin, and no errors are reported if the lockset becomes empty unless the variable has been written to. This eliminates false errors reported for read–only data. Eraser was designed to operate on a range of synchronization mechanisms and has extra complexity to deal with read–write locks (multiple readers but only one writer). Since Java does not have a standard concept of such locks, we omit this complexity from the algorithm.

Eraser's heuristics to deal with its relaxed discipline can cause it to miss violations

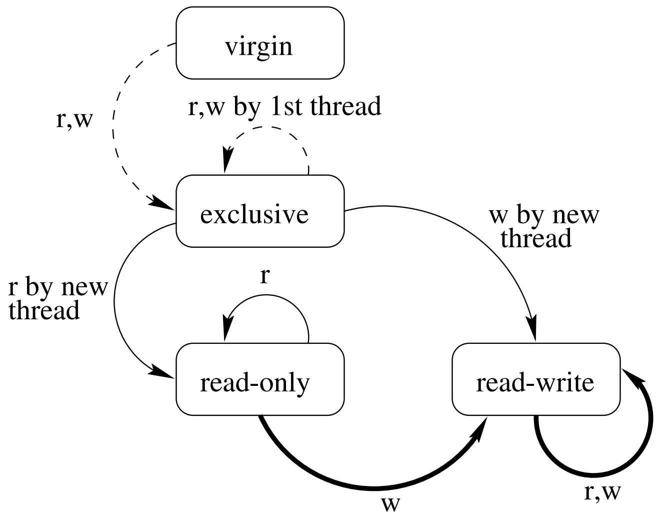

Figure 2.1: State diagram of the state that Eraser maintains for each variable. Transitions are performed on reads (indicated by "r") and writes (indicated by "w") to the variable. A dashed line indicates that no lockset refinement is performed. A thin solid line indicates that lockset refinement is performed but that no violations are reported if the lockset becomes empty. A thick solid line indicates that lockset refinement is performed and an error is reported if the lockset becomes empty. (This figure is adapted from [S+97].)

of the locking discipline. Savage et al. argue that their empirical evidence from a variety of actual systems bears out their hypothesis that these heuristics do not lead to missing actual data races or mutual–exclusion errors in practice, and prevent reporting numerous discipline violations that are not really program errors. Our tester works properly for this relaxed discipline.

Eraser by itself cannot make any guarantees that it finds all data races present in a program. Consider the code in Figure 2.2: Eraser will detect the data race in this program in only some of the possible schedules.

2.1.2 Finalization

Classes in Java may have `finalize` methods, or *finalizers*, that are called by the garbage collector when instances of those classes are reclaimed. There are no guarantees on when finalizers are called. They can be called in any order or even not at all, depending on when or if the garbage collector runs. This means that the tester would need to execute every possible schedule of finalizers with the rest of the program in order to find possible assertion violations, which can be a very large number of schedules. However, most finalizers are

```java
public class NoEraser implements Runnable {
  static class Resource { public int x; }
  static Resource resource = new Resource();

  // Two threads A and B
  // iff B writes resource.x before A -> Eraser data race
  public static void main(String[] args) {
    new NoEraser("A");
    new NoEraser("B");
  }

  private String name;

  public NoEraser(String name) {
    this.name = name;
    new Thread(this).start();
  }

  public void run() {
    int y;
    synchronized(resource) {
      y = resource.x;
    }
    if (y==0 && name.equals("B"))
      resource.x++;
    else synchronized(resource) {
      resource.x++;
    }
  }
}
```

Figure 2.2: A program that will pass Eraser's test nondeterministically. If the thread named `B` increments `resource.x` before `A`, `B` does so without obtaining the lock for `resource` first. This is a data race that Eraser will detect. However, if `A` increments `resource.x` before `B` does, Eraser will not report any errors.

used only to deallocate memory that was allocated in some native Java method or system resources such as open file descriptors. Since these activities do not interact with the rest of the program, the order of execution of such finalizers does not affect the rest of the program. To reduce the number of schedules tested, the tester assumes that the timing of finalizers has no bearing on whether assertions will be violated or not or whether a deadlock can occur. We call this the tester's *finalization criterion.*

This criterion is met when a program's finalizers access only variables of `this` (the instance object they are called on) and native state that does not affect the rest of the program. In order to ensure that no deadlocks can occur due to the timing of a finalizer, finalizers should not contain nested synchronized regions or perform `wait` or `notify` operations.

The finalization criterion seems to be quite reasonable and is met by the finalizers in the standard Java libraries. The asynchronous timing of finalizers make it difficult for them to correctly interact with the rest of the program, so programmers tend not to violate the criterion. The tester could check that the finalization criterion is met by examining all finalizers in a program for these properties, but we have not attempted this.

Given this assumption, the tester can ignore the garbage collector completely, since the only way it can affect whether assertions are violated or not is through the timing of finalizers. The tester makes no guarantees about programs that do not meet the finalization criterion.

2.1.3 Terminating Threads

The tester requires that all of the threads that compose a program it is testing are terminating. This is so that the depth–first search of the program's schedules that the tester performs will terminate. If a program has non–terminating threads, the user can either modify the source code (for example, changing an infinite loop to an n–iteration loop for some n) or the tester can limit each thread to execute a maximum of n bytecodes. In this case the tester would only guarantee to find all assertion violations and deadlocks in schedules involving n or fewer bytecodes executed in each thread. Another solution might be to have the tester stop threads when they leave a specified module of interest.

The tester could check that a program's threads are terminating by having a default value for n; when it finds a thread that runs for more than n bytecodes it would notify the user that the thread executed for too long and that its termination should be examined.

2.2 Testing Algorithms Overview

A program following a mutual–exclusion locking discipline can be divided into *atomic blocks* based on blocks of code that are protected by synchronized statements. Shared variables cannot be accessed outside of synchronized atomic blocks. Furthermore, shared variables modified by a thread inside of an atomic block cannot be accessed by other threads until the original thread exits that block. This means that enumerating possible orders of the atomic blocks of a program covers all possible behaviors of the program.

To see why enumerating the orders of the atomic blocks is sufficient, consider the following. The order of two instructions in different threads can only affect the behavior of the program if the instructions access some common variable — one instruction must write to a variable that the other reads or writes. In a program that correctly follows a mutual–exclusion locking discipline, there must be at least one lock that is always held when either instruction is executed. Thus they cannot execute simultaneously, and their order is equivalent to the order of their synchronized regions. By enumerating only synchronized regions, we significantly reduce the number of schedules to consider — rather than considering all possible schedules at the instruction level, the tester need only consider schedules of the program's atomic blocks.

Since the atomic blocks of a program can change from one execution to the next due to data flow through branches, we cannot statically compute a list of atomic blocks. We instead dynamically enumerate the atomic blocks using depth–first search. Depth–first search requires that the threads of the program are terminating; this is the reason for the criterion that all threads being tested are terminating.

To perform depth–first search on a program we first execute one complete schedule of the program. Then we back up from the end of the program to the last atomic block boundary at which we can choose to schedule a different thread. We create a new schedule, branching off from this point in the program by choosing a different thread to run. We again execute until program completion. Then we systematically repeat the process, continuing to back up to atomic block boundaries where we can make different choices than before.

We end up with a tree of schedules. Figure 2.4 illustrates a sample tree for the three

```
Thread 1 =
A:   synchronized (x) { <arbitrary code> }

Thread 2 =
B:   synchronized (x) { <arbitrary code> }

Thread 3 =
C:   synchronized (x) { <arbitrary code> }
```

Figure 2.3: Three threads that each contain a single synchronized region, and thus a single atomic block. The sections marked <arbitrary code> indicate regions of code that do not contain synchronized statements.

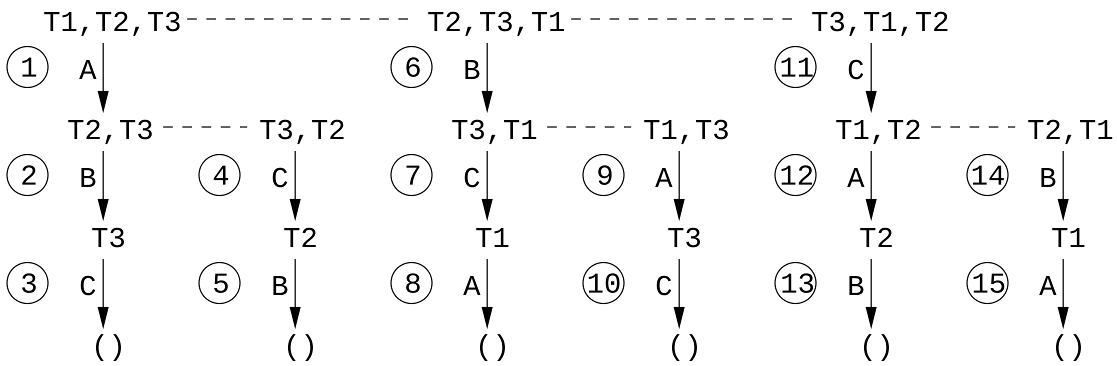

Figure 2.4: Tree of schedules for the three threads of Figure 2.3. At each node of the tree we list the live threads; the first thread listed is the one that we chose to execute from that node. An arrow indicates the execution of a single atomic block, labeled on its left. A horizontal dashed line indicates that the two nodes it joins are equivalent client program states from which the two subtrees follow different schedules. The circled numbers show the order in which the atomic blocks are executed by our depth–first search.

single–atomic–block threads of Figure 2.3. The circled numbers show the order in which the atomic blocks are executed by the search, and dashed lines indicate points where we branch off new schedules. This tree represents all possible schedules of the three atomic blocks: ABC, ACB, BCA, BAC, CAB, and CBA.

To implement this search we need the ability to back up the program to a previous state, and the ability to present a consistent external view of the world to the program as we execute sections of code multiple times. Our tester uses checkpointing to back up and deterministic replay to ensure the program receives the same inputs as it is re–executed; the implementation of these is discussed in Chapter 3.

This method will find an assertion violation in a program if one can occur, regardless of whether the violation shows up when the program is executed on a single processor or a multiprocessor. The locking discipline enforces an ordering on shared–variable accesses — two threads executing on different processors cannot access the same variable at the same time. Theorem 1 in Section 2.3.3 proves that the execution of a program following the discipline on a multiprocessor is equivalent to some serialization of that program on a single processor.

The ExitBlock algorithm presented in the next section systematically explores schedules of atomic blocks by dynamically exploring the tree of schedules using depth–first search. The number of schedules is reduced further in the ExitBlock-RW algorithm of Section 2.4, which uses data dependencies between atomic blocks to identify which sets of blocks' orderings do not affect the program's behavior. The final algorithms presented (in Section 2.5) are the ExitBlock-DD and ExitBlock-RWDD algorithms, which detect deadlocks without executing any more schedules than ExitBlock or ExitBlock-RW, respectively.

2.3 The ExitBlock Algorithm

The assumptions discussed in Section 2.1 allow the tester to systematically explore a program's behaviors by considering only schedules of atomic blocks. The tester also needs to consider each possible thread woken by a `notify` operation. The extra work needed to deal with `notify` is discussed in Section 2.3.1; we ignore it for now.

For un–nested synchronized regions, it is clear what an atomic block is: the region from a lock enter to that lock's exit (a lock enter is an acquisition of a lock, which occurs at the beginning of a synchronized block or method, while a lock exit is the release of a lock that occurs at the end of a synchronized block or method and in `wait` operations). The first atomic block of a thread begins with its birth and ends with its first lock exit, while the last atomic block begins with its last lock exit and ends with its death.

For the sample thread in Figure 2.5, the code sections labeled 2 and 4 must be in separate atomic blocks. The rest of the sections can be grouped in any manner; we could have two blocks, 1,2,3 and 4,5, or 1,2 and 3,4,5, or we could have three blocks, 1,2, 3,4, and 5, or many other combinations. We want as few blocks as possible, so we choose to group the non–synchronized code before a synchronized region with that region in one atomic block. We would like to join code ending in thread death with the previous atomic block,

```
Thread 1 =
1:   <arbitrary code>
2:   synchronized (a) { <arbitrary code> }
3:   <arbitrary code>
4:   synchronized (b) { <arbitrary code> }
5:   <arbitrary code>
```

Figure 2.5: An example thread that has two synchronized regions. The sections marked `<arbitrary code>` indicate regions of code that do not contain `synchronized` statements. The code is arranged with entire regions falling on single lines to facilitate referring to regions by number. We divide the thread into three atomic blocks: 1,2, 3,4, and 5.

```
Thread 1 =
1:   <arbitrary code>
2:   synchronized (a) { <arbitrary code>
3:       synchronized (b) { <arbitrary code> }
4:       <arbitrary code> }
5:   <arbitrary code>

Thread 2 =
6:   <arbitrary code>
7:   synchronized (a) { <arbitrary code> }
8:   <arbitrary code>
9:   synchronized (b) { <arbitrary code> }
10:  <arbitrary code>
```

Figure 2.6: Two threads, one containing nested synchronized regions. Even for nesting we delineate atomic blocks with lock exits. We divide `Thread 1` into atomic blocks 1,2,3, 4, and 5, and `Thread 2` into atomic blocks 6,7, 8,9, and 10.

but there is no way to know beforehand when a thread is going to die, so we keep them separate. Thus we have three atomic blocks: 1,2, 3,4, and 5.

Now we need to deal with nested synchronized regions. Perhaps we can generate all program behaviors by focusing on synchronized regions of one lock variable at a time. This way we never encounter any nesting. If a program has k lock variables, and we make k passes through the program, each time considering schedules of atomic blocks defined only by the synchronized regions of the kth lock variable, will the sum of those k passes cover all behaviors of the program? The answer is no. This is because shared variables are typically not completely independent of one another in their effect on program behavior. Information from two shared variables that are protected by different locks can be combined (for example, into a local variable that holds their sum). We need to consider atomic blocks based on all synchronized regions.

We must decide how to divide nested synchronized regions into atomic blocks. As an example, consider the code for `Thread 1` in Figure 2.6. Should the code section labeled 2 be in a separate atomic block from section 3? Or can we combine them? For the purpose of finding all assertion violations, it turns out that we *can* combine them. This may seem counter–intuitive. Intuition suggests that we need to schedule `Thread 2`'s section 9 in between sections 2 and 3, because `Thread 2` could modify variables protected by the `b` lock while `Thread 1` is in section 2. However, this is the same as having section 9 execute before both sections 2 and 3. Intuition also suggests that we should schedule section 7, 8, and 9 in between sections 2 and 3; this cannot occur, however, because while `Thread 1` holds the `a` lock section 7 cannot execute. We thus define atomic blocks to be sections of code in between lock exits (thread birth and death also define borders of atomic blocks, of course).

Ignoring `wait` and `notify` for the moment, the ExitBlock algorithm is shown in Figure 2.7; it systematically executes the schedules of atomic blocks delineated by lock exits. ExitBlock dynamically explores the tree of possible schedules using depth–first search. Each thread in the program is kept in one of three places:

1. In the "block set" of a lock, if the thread needs that lock to continue its execution and another thread holds the lock.

2. As the "delayed thread" — the thread that is not allowed to execute at this time in order to schedule another thread.

3. In the "enabled set", the set of threads that are ready to be executed.

ExitBlock runs the program normally until there are two threads; then it places both of them in the enabled set and enters its main loop. Each iteration of the loop is the execution of an atomic block of the current thread, which is chosen from the enabled set. First a checkpoint is taken and the enabled set saved; then the current thread is chosen and executed. When it reaches a lock exit, a new branch of the tree of schedules is created and stored on a stack for later execution (remember, ExitBlock uses depth–first search). This branch represents executing the rest of the threads from the point just prior to this atomic block. We need the checkpoint so we can go back in time to before the block, and we need to make sure we execute a different thread from that point. So we make the current thread the "delayed thread" of the branch. The delayed thread is re–enabled after the first atomic

```
BEGIN:
 run program normally until 2nd thread is started, then:
     enabled = {1st thread, 2nd thread}
     delayed_thread = null
     goto LOOP
LOOP:
 if (enabled is empty)
     if (stack is empty) goto DONE
     pop (RetPt, saved_enabled, saved_thread) off of stack
     change execution state back to RetPt
     enabled = saved_enabled
     delayed_thread = saved_thread
     goto LOOP
 curThread = choose a member of enabled
 RetPt = make new checkpoint
 old_enabled = enabled
 run curThread until one of the following events occurs:
     if (lock_enter && another thread holds that lock)
         move curThread from enabled to lock's block set
         change execution state back to RetPt
         goto LOOP
     if (curThread dies or reaches execution limit)
         remove curThread from both enabled and old_enabled
         goto LOOP
     if (thread T starts)
         add T to enabled set
         continue running curThread
     if (lock_exit)
         save_enabled = old_enabled minus curThread
         push on stack (RetPt, save_enabled, curThread)
         add delayed_thread to enabled set
         add all threads on lock's blocked set to enabled set
         goto LOOP
DONE: // testing is complete!
```

Figure 2.7: Pseudocode for initial ExitBlock algorithm that does not handle `wait` or `notify`. ExitBlock systematically executes the tree of possible schedules by dynamically discovering the atomic blocks of a program. At each lock exit (atomic blocks are delineated by lock exits) it pushes another branch of the tree onto its stack to be executed later.

block of a branch so that it can be interleaved with the atomic blocks of the other threads. Note that we use the old enabled threads set in the branch; this is because newly created threads cannot interact with their own atomic block, and in fact did not exist at the point of the checkpoint.

ExitBlock keeps executing the current thread, pushing branches onto the stack, until the thread dies. Then it chooses a different thread from the enabled set to be the current thread and executes it in the same manner. ExitBlock treats the execution from a thread's final lock exit to its death as a separate atomic block; this is unavoidable since we have no way of knowing when a thread will die beforehand. When the enabled set becomes empty ExitBlock pops a new branch off of the stack and uses the branch's state as it continues the loop.

If the current thread cannot obtain a lock, a new thread must be scheduled instead. Since we only want thread switches to occur on atomic block boundaries, we abort the current branch of the tree and undo back to the end of the previous atomic block before we schedule a different thread. Which threads own which locks must be kept track of to determine if a lock can be acquired without having the thread actually block on it. The thread is placed in the block set of the lock and a different enabled thread is tried after returning to the checkpoint. ExitBlock cannot enter a lock–cycle deadlock since it never lets threads actually block on locks, so that is not a concern (see Section 2.5).

ExitBlock assumes that it has the power to take and return to checkpoints, and to deterministically replay all interactions between the Java code of the program and the rest of the world: input, output, and native methods. It must do so to make the threads of the program deterministic with respect to everything except variables shared with other threads. The pseudocode in this chapter assumes that deterministic replay is going on behind the scenes; Section 3.3.2 discusses how to implement deterministic replay.

Figure 2.8 shows the tree of schedules executed by ExitBlock for a program consisting of the two threads in Figure 2.6. `Thread 1` is abbreviated as `T1` and `Thread 2` is abbreviated as `T2` in the figure. Each node lists the threads in the enabled set, and the delayed thread, if there is one, in parentheses. An arrow indicates execution of one atomic block, with the section numbers of the code contained in the block listed to the left of the arrow. Parallel execution paths are connected by dashed lines to indicate that they are both executed from a checkpoint that was created at the left end of the dashed line. The large `X` in the figure

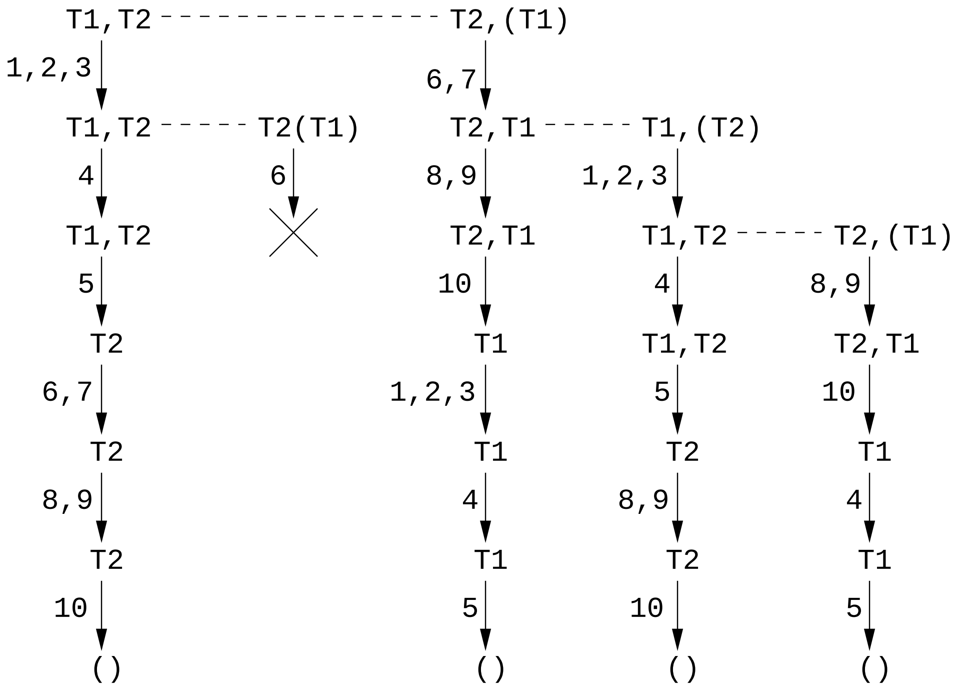

Figure 2.8: Tree of schedules explored by ExitBlock for the threads in Figure 2.6. Threads in parentheses are delayed. Arrows indicate the execution of the sections of code on their left. An `X` means that a path was aborted because a lock could not be obtained.

indicates that that path was aborted because `T2` could not obtain a lock that `T1` was holding. A checkpoint is created at each node in the tree, but most are not used.

2.3.1 Thread Operations

So far we have ignored thread communication other than with shared variables protected by locks; in particular, we have ignored `wait` and `notify`. A thread can `wait` on an object, causing the thread to go to sleep until another thread performs a `notify` (awakening one thread) or `notifyAll` (awakening all threads) on that object. We can deal with `wait` by considering the lock exits that result when a thread `wait`s to be like other lock exits. A thread that has acquired a lock multiple times, which is permitted by Java, will release it multiple times when it performs a `wait` on the lock object. Since there is no way to schedule another thread in between these releases, we consider them to be one lock exit. Also, a thread that performs a `wait` should be removed from the set of enabled threads for the duration of its `wait`.

Dealing with the notification operations is more involved. For `notifyAll` we need to add every thread that is woken up onto the block set for the notify lock. For `notify` we need to do the same thing for the single thread that wakes up; however, if there are multiple threads waiting on an object, which thread will be woken by `notify` is nondeterministic. The tester needs to explore the schedules resulting from each possible thread being woken by a `notify`, so it needs to make new branches for each. Figure 2.9 shows additions to the ExitBlock algorithm that will handle `wait` and `notify`. For each branch of the tree in which we wish to notify a different thread, we must prevent threads that have already been notified from being notified again. Thus we keep track of a set of threads that should not be notified (the "no_notify" set) for every branch of the execution tree. Even if there are multiple notifications in one atomic block, we can still use a single no_notify set since there can be no overlap between threads waiting on the multiple notify objects (a thread can only `wait` on one object at a time). When we execute a `notify` we wake up a thread that is not in the no_notify set. We want to carry the set to the next branch's identical `notify` operation in order to keep reducing the set of threads we wake up; however, we do not want a different `notify` that occurs later to have the threads it wakes up restricted by this `notify`. Therefore, after pushing a new `notify` branch on the stack, we remove from the no_notify set all threads originally waiting on the notify object.

The other thread–related operations are simpler to handle:

- `sleep` — we are already considering all behaviors assuming threads can be preempted at any time, so a thread switch resulting from a `sleep` call is already being considered.
- `yield` — we are already considering preemption everywhere, so a yield changes nothing in the schedules we consider. Furthermore, we do not want to execute the `yield` since we want to thread switch only on lock exits.
- preemption — again, we are already considering preemption everywhere. As with yield, we do not want any thread switches except at lock exits.
- `setPriority` — we must ignore priorities, since Section 17.12 of [GJS96] does not guarantee that a lower priority thread cannot preempt a higher. We have to consider schedules of threads regardless of priority.
- `stop` — we wait until the stopped thread really dies, and ExitBlock already deals with thread death.

```
BEGIN:
 run program normally until 2nd thread is started, then:
     wait = {}
     no_notify = {}
     ...
LOOP:
 if (enabled is empty)
     ...
     pop (RetPt, saved_enabled, saved_thread, wait,
         no_notify) off of stack
     change execution state back to RetPt
     ...
 ...
 run curThread until one of the following events occurs:
     ...
     if (curThread does a wait())
         move curThread from enabled set to wait set
         count as lock_exit
     if (curThread does a notifyAll())
         wake up all threads waiting on the notify object
         move those threads from wait set to block set of notify lock
     if (curThread does a notify())
         select a thread T not in no_notify to wake up
         move T from wait set to block set of notify lock
         if (there are still threads waiting on the notify object)
             push on stack (RetPt, enabled, delayed_thread, wait,
                 no_notify+T)
         no_notify = no_notify -
                     all threads originally waiting on notify object
     if (lock_exit)
         ...
         push on stack (RetPt, save_enabled, curThread, wait, {})
         ...
DONE: // testing is complete!
```

Figure 2.9: Changes to the pseudocode for the ExitBlock algorithm in Figure 2.7 in order to handle `wait` and `notify`. A set of waiting threads and a set of threads that should not be notified are added to each branch. We treat `wait` like a lock exit, and for `notify` we create new branches of the tree to consider other threads being notified.

- `join` — turns into `wait`.

- blocking i/o operation — since we are testing the program for one particular input, we assume the user will provide any needed input and that blocking on an input will not be a problem; replaying the input deterministically for other paths is necessary and, as discussed earlier, is assumed to be happening all along. Implementing this deterministic replay is discussed in Section 3.3.2.

- `suspend` and `resume` — the tester needs to ensure that a suspended thread is not in the enabled set, and that a non–suspended thread is. A resumed thread can still be waiting or blocked on a lock, so care must be taken on adding it to the enabled set. The details of this have not been completely worked out, so the pseudocode ignores these thread operations.

2.3.2 Number of Schedules Executed by ExitBlock

For a program with k threads that each contain n lock exits (say, m different locks obtained $\frac{n}{m}$ times each), the total number of lock exits we have is $k * n$. $\binom{kn}{n}$ is the number of ways we can place the first thread's n lock exits in between all of the other lock exits. Placing a lock exit also places the atomic block that the lock exit terminates. Now that those are placed, we have $(k-1)*n$ slots left; $\binom{(k-1)n}{n}$ is the number of ways the second thread can place its atomic blocks in between the others. The process continues until we have the number of schedules that the ExitBlock algorithm needs to consider:

$$\binom{kn}{n} * \binom{(k-1)n}{n} * \binom{(k-2)n}{n} * \cdots * \binom{n}{n} \tag{2.1}$$

Stirling's approximation gives us

$$\binom{n}{r} \sim \frac{1}{\sqrt{2\pi}} \frac{\sqrt{n}}{\sqrt{r(n-r)}} \left(\frac{r}{n}\right)^{r} \left(\frac{n}{n-r}\right)^{n-r}$$

so we have

$$\binom{kn}{n} = \theta\left(\frac{\sqrt{k}}{\sqrt{n(k-1)}} \left(\frac{1}{k}\right)^{n} \left(\frac{k}{k-1}\right)^{n(k-1)}\right)$$

Equation 2.1 contains a product of exponential terms; the result is still exponential. Thus the number of schedules that must be explored grows exponentially in both the number of threads and the number of lock uses by each thread.

How does this compare with an algorithm that did not assume a mutual–exclusion locking discipline and simply interleaved every atomic instruction? If we assume that there are an average of 100 instructions per lock exit, we can simply replace n with $100n$ in our formulas. This is also exponential growth, but much faster exponential growth. If we fix $k = 2$ we have, for the first term of Equation 2.1,

$$\binom{2n}{n} = \theta\left(\frac{2^{2n}}{\sqrt{n}}\right)$$

for ExitBlock versus

$$\binom{200n}{100n} = \theta\left(\frac{2^{200n}}{\sqrt{n}}\right)$$

for an instruction–interleaving algorithm. This difference is huge. Of course, ExitBlock's paths are still growing exponentially. Section 2.4 will modify the ExitBlock algorithm to reduce the best–case growth of the number of paths from exponential to polynomial in the number of locks per thread.

As can be seen, most real concurrent programs will have too many schedules to search in a reasonable amount of time (a few hours). The tester could explore portions of programs by executing the program normally (i.e., nondeterministically, without enumerating all behaviors) up to a certain point and from that point on systematically exploring behaviors. We plan to embed the tester in a sophisticated program development environment and in conjunction with a debugger and other tools that will make use of its systematic search abilities within focused contexts.

Another planned feature of the tester to cope with long execution times is to record traces of its execution and replay particular schedules. For example, it will record all schedules that lead to assertion failures during an overnight run so that those schedules can be examined in detail by the user in the morning.

2.3.3 Correctness Proof for ExitBlock

First a few notes:

- A mutual–exclusion locking discipline does not distinguish between reads and writes — all accesses to a variable must be protected by the same set of locks. This means that we only have one kind of dependency between variables, access dependencies, rather than the three kinds of dependency we would have to consider if reads and writes were treated differently (namely, write-read dependencies, write-write dependencies, and read-write anti-dependencies).

- Java locks are owned by threads. If one thread obtains a lock, only that thread can subsequently release the lock.
- Java enforces properly paired and nested lock enters and exits. This means that the locks owned by a thread when it acquires lock `L` cannot be released until after it releases `L`. There are no separate lock enter and exit language mechanisms, only synchronized blocks.
- A thread cannot die without first releasing all locks that it holds. This follows from Section 20.20 of [GJS96], which lists the ways in which a thread can die:
  - Its run method completes normally. Since a thread cannot hold locks before its run methods begins, it cannot be holding any locks after the run method returns, because Java constrains lock enters and exits to be properly paired.
  - Its run method completes abnormally, i.e., an exception is thrown. By Section 14.17 of [GJS96], if execution of a synchronized block "completes abruptly for any reason, then the lock is unlocked." And by Section 8.4.3.5 of [GJS96], a synchronized method acts just like it was a normal method whose body is wrapped in a synchronized block. Thus after abnormal completion of the run method no locks may be held.
  - The thread invokes `Thread.stop`, or some other thread invokes `stop` on this thread: `Thread.stop` always results in some `Throwable` being thrown, which by the reasoning above will cause all held locks to be released.
  - `System.exit` is invoked. In this case the entire program terminates, so we are not concerned with who holds what locks.

And a few definitions:

Definition 1 *An* atomic instruction *is the smallest execution fragment that cannot have a thread switch occur in the middle of it (when executed on a single processor).*

Opcodes that only affect local state, such as `aload` or `iinc`, are atomic in this sense. For instructions that can affect non-local memory, we turn to the definition of thread *actions* in Section 17 of [GJS96]. Opcodes like `putfield` and `getfield` are not atomic but are made up of several actions; for example, `getfield` consists of the actions `read`, `load`, and `use`. These actions are our atomic instructions.

A thread issues a stream of atomic instructions. Section 17 of [GJS96] states that the order of instructions can be viewed by another thread as being different from the order in which they are issued (in particular, writes need not be completed in order); however, since we are only considering programs that follow a mutual–exclusion locking discipline, and since a lock exit forces all pending writes to be completed, the order in which a thread issues its instructions can be considered to be the order that other threads view those instructions without loss of generality.

Definition 2 *A* schedule *S of a program P is a representation of a particular execution of P on a single processor that denotes the interleaving of execution of the threads of P on that processor. Let $T = \{t_1, \ldots, t_n\}$ be the set of threads that P started during its execution. Each thread t_i is composed of the sequence of m_i atomic instructions that it issued during P's execution: $t_i = a_{i1} \ldots a_{im_i}$. Now $S = s_1 s_2 \ldots s_n$, where each s_i is a maximal non–empty sequence of atomic instructions $a_{\alpha 1} \ldots a_{\alpha k}$, $\alpha \in \{1, \ldots, n\}$ called a* segment *of thread t_α. Every thread in T must be represented in S in its entirety with its instructions in order. So for each $t_i \in T$ let $u_i = \{s_i \in S \mid s_i \text{ is a segment of } t_i\}$. Now if we arrange the elements of u_i in order (so that s_i precedes s_j if $i < j$) and expand each s_i into its constituent atomic instructions, the resulting sequence of atomic instructions must equal the sequence of atomic instructions that makes up t_i. There is one final requirement: in a schedule a thread segment cannot appear before that thread is created by some other thread's segment (with the exception of the initial thread, of course).*

As explained in Section 2.1.2, we ignore the garbage collector and finalization of objects. Thus we do not consider any garbage collection threads or finalization threads in our schedules. They are simply not present.

Definition 3 *A* lock exit *is either a monitorexit, a return from a synchronized method, or the lock release that occurs when a* `wait` *operation is performed. A lock exit is equivalent to an* `unlock` *action of Section 17 of [GJS96].*

Definition 4 *An* exit–block schedule *is a schedule in which all segments end with lock exits or thread deaths. Note that Java requires that a thread that dies call* `notifyAll` *on* *itself, which involves acquiring the lock for the thread object. This means that no thread can execute, however briefly, without its execution containing at least one lock exit.*

Now we prove that ExitBlock finds all assertion violations. First we show that it does not matter whether a program is executed on a multiprocessor or a single processor:

Theorem 1 *The behavior of a Java program that follows a mutual–exclusion locking discipline on a multiprocessor machine is equivalent to some serialization of that program on a single processor.*

Proof: Let P be the program and $Q = \{q_1, \ldots, q_n\}$ be the set of processors on the multiprocessor machine. Let $S_i = s_{i1}s_{i2}\ldots s_{im_i}$ be the schedule executed by processor q_i. Now serialize the S_i into a single schedule S as follows:

- If s_{ij} began and completed before s_{kl} began (in real time), place s_{ij} before s_{kl} in S.
- If s_{ij}'s execution overlapped with s_{kl}'s execution, split both segments into their atomic instructions and place those instructions into S in the order that they executed. If any of those instructions executed simultaneously, place them in S in arbitrary order.

Why can simultaneously executed instructions be placed in arbitrary order? Their order can only matter if there are dependencies between them. Given that the rest of S is in the order of real time execution, there can only be dependencies between them if one or more of them reference variables that are shared between their respective threads. Now, since the program follows a mutual–exclusion locking discipline, only one thread at a time may access shared variables. Thus simultaneously executed instructions cannot have any dependencies between them and placing them in S in arbitrary order will have no effect on the execution of the program.

Therefore this serialization produces a schedule S whose execution on a single processor will be indistinguishable from the execution of P on Q. $\square$

Next we show that ExitBlock executes all exit–block schedules of a program. Finally, we show that an assertion violation that occurs in some execution of a program must also occur in one of the exit–block schedules. In the remaining theorems, for simplicity, we shall only consider programs executed on a single processor. Theorem 1 allows these theorems to apply to programs executed on any number of processors.

Theorem 2 *Consider a program P that meets the testing criteria defined in Section 2.1. For a given input, let S_{all} be the set of all possible schedules of P when executed on a single processor. Let S_{exit} be the set of exit–block schedules in S_{all}. Then the ExitBlock algorithm run on P for the same input executes every schedule in S_{exit}.*

Proof: By contradiction. Assume that an exit–block schedule $S = s_1 s_2 \ldots s_m$ that is a member of S_{exit} exists that the ExitBlock algorithm does not execute. Compare S to all schedules executed by ExitBlock and find the longest prefix of S that matches a schedule executed by ExitBlock. Let s_n be the first segment in which S differs from that schedule. We have $0 \leq n \leq m$ such that $s_1 \ldots s_{n-1}$ is a prefix of some schedule produced by ExitBlock, but s_n is never executed directly after s_{n-1} by ExitBlock.

Let t be the thread of which s_n is a segment. t must have been created earlier than s_n, and it must not be blocked on a lock or waiting after s_{n-1} since S is a valid schedule. t cannot be delayed either. A thread is only delayed for the first atomic block of a new branch; the new branch is created after executing the next atomic block of the delayed thread. So if t were delayed after s_{n-1} then a new branch must have been created in response to t executing the first atomic block of s_n. But if s_n's first atomic block was executed, then the rest of s_n must have been executed as well, for ExitBlock lets the currently executing thread continue until it dies, performs a `wait`, or blocks on a lock. Since S exists, we know that t does none of these things in the segment s_n. Thus t cannot be delayed without this branch's creator having executed s_n immediately after s_{n-1}, which would contradict our assumption. This means that t must be in ExitBlock's enabled set at the end of s_{n-1}.

The ExitBlock algorithm then executes the next segment of some thread in its enabled set. If it chooses thread $t' \neq t$, it will (after the first lock exit) push onto its stack a new branch. This branch's task is executing all threads that were enabled at the end of s_{n-1} (except for t') from the checkpoint taken immediately after s_{n-1}. Because ExitBlock does not terminate until the stack is empty, it will eventually execute the stored branch. If it does not choose to execute t then, it will store another branch with a smaller enabled set for execution from the same checkpoint. Stored branches with still smaller enabled sets will be created until the enabled set is empty. Thus ExitBlock will eventually choose to execute t from the checkpoint after s_{n-1}.

Once ExitBlock executes the next segment with t, it will keep executing t (pushing alternative branches onto the stack along the way) until t dies, performs a `wait`, or blocks on a lock. Because S exists, t must not die, `wait`, or block on a lock during the code executed by it in s_n. Thus ExitBlock will execute t at least until the end of the code in s_n.

If there are no other enabled threads at this point, then $n = m$ and we have shown that ExitBlock executes S and have reached a contradiction. If there are, then since s_n ends in a lock exit or a thread death (otherwise it could not be a segment) ExitBlock will create a branch for execution of some other thread from a checkpoint immediately after s_n. This means that ExitBlock will executed a schedule with a prefix $s_1 \ldots s_n$. But this is also a contradiction; therefore the assumption that S exists must be incorrect. This proves the theorem. □

Theorem 3 *Consider a program P that meets the testing criteria defined in Section 2.1. Suppose that for a given input, when executed on a single processor, P can produce an assertion violation. Consider an execution of P that produces the violation. Let $T = \{t_1, \ldots, t_n\}$ be the set of threads that P starts during this execution, and let $S = s_1 s_2 \ldots s_m$ be the schedule of these threads that leads to the assertion violation. A schedule S' exists that is produced by the ExitBlock algorithm in which the same assertion is violated.*

Proof: We construct S' from S as follows. For each s_i we apply the following transformations. Let $t \in T$ be the thread executing s_i.

- If s_i contains no lock exits, move all of the atomic instructions from s_i forward to the beginning of the nearest s_j, $j > i$, such that s_j's thread is also t. If such an s_j does not exist then s_i must end in a thread death. In this case, move all of the atomic instructions from s_i backward to the end of the nearest s_k such that s_k's thread is also t. Neither of these movements affects whether the assertion is violated or not, as argued below:

    1. In the case that s_j exists, the movement of s_i obviously does not change the execution of any bytecodes prior to s_i. It also does not change the execution of any bytecodes between s_i and s_j (and consequently no bytecodes from s_j onward). This follows from the mutual–exclusion locking discipline: if s_i accesses any shared variables, t must be holding the set of locks associated with each of those variables. Because s_i performs no lock exits, and because no other thread can release any locks for t (Section 17.5 of [GJS96]), those locks are still held at

least until the next segment of t, which is s_j. Thus no intervening segments may obtain those locks in order to access any shared variables that s_i accesses. With no dependencies between s_i and the intervening code, moving s_i to the beginning of s_j does not affect whether or not the assertion is violated.

2. In the case that s_j does not exist, the movement of s_i obviously does not change the execution of any bytecodes prior to s_k. It also does not change the execution of any bytecodes between s_k and s_i (and consequently no bytecodes from s_i onward). This follows from the mutual–exclusion locking discipline: if s_i accesses any shared variables, t must be holding the set of locks associated with each of those variables. Because s_i contains no lock exits and a thread cannot die while holding a lock (as shown earlier), t must hold no locks at all during s_i. This means that there can be no shared variables accessed by s_i. With no dependencies between s_i and the intervening code, moving s_i to the end of s_k does not affect whether or not the assertion is violated.

- If s_i contains lock exits, split s_i into two segments, the first from the beginning of s_i until its last lock exit, and the second from after its last lock exit until the end of s_i. Splitting it into two segments is a valid operation because a thread switch to the currently running thread does not change the execution in any way. Now the second of the newly created segments falls under the first case above. We leave the first segment alone.

We now have an S' that executes the same instructions as S. Each transformation does not affect the condition that triggered the assertion violation; each instruction of S' receives the same inputs as the corresponding instruction in S. Thus the condition that triggered the assertion violation will be reached in S' just as it is in S.

S' is an exit–block schedule. This follows from the process by which S' was created from S: S' consists of segments that all end with lock exits, except for those that end in thread deaths. By Theorem 2, ExitBlock produces all exit–block schedules of P. Thus the ExitBlock algorithm produces S', a schedule in which the same assertion is violated as in S. $\square$

2.4 The ExitBlock-RW Algorithm

The ExitBlock algorithm executes all schedules of atomic blocks. However, all schedules need not be executed in order to find all possible assertion violations. If two atomic blocks have no data dependencies between them, then the order of their execution with respect to each other has no effect on whether an assertion is violated or not. The ExitBlock-RW algorithm uses data dependency analysis to prune the tree of schedules explored by ExitBlock.

As an example, consider again the tree from Figure 2.8, which is repeated here as Figure 2.10. The rightmost branch represents the ordering `6,7,1,2,3,8,9,10,4,5`. Code sections 4 and 5 only share locks with section 7 of the other thread. This means that 4 and 5 can have data dependencies only with 7 and with no other code sections of the other thread. We have already considered 4 and 5 both before and after 7 in earlier branches. So why should we execute 4 and 5 at the end of this last branch? If there were an assertion violation produced only when 4 and 5 were executed after 7, we would already have found it. Thus we can trim the end of the rightmost branch.

We take advantage of this observation as follows. We record the reads and writes performed while executing an atomic block. Instead of delaying just the current thread and re–enabling it after the first step in the new branch, we keep a set of delayed threads along with the reads and writes of the atomic block they performed just before the new branch was created. We only re–enable a delayed thread when the currently executing thread's reads and writes intersect with the delayed thread's reads and writes. If no such intersection occurs, then none of the later threads interact with the delayed thread and there is no reason to execute schedules in which the delayed thread follows them. The intersection is computed as follows: a pair consisting of a set of reads and a set of writes (r_1, w_1) intersects with a second pair (r_2, w_2) if and only if $(w_1 \cap w_2 \neq \emptyset \vee r_1 \cap w_2 \neq \emptyset \vee w_1 \cap r_2 \neq \emptyset)$.

We call the ExitBlock algorithm augmented with this read–write pruning ExitBlock-RW. Pseudocode for the changes to ExitBlock required for ExitBlock-RW is given in Figure 2.12. Note that the algorithm uses the reads and writes performed during the first execution of an atomic block A to check for data dependencies with other threads' atomic blocks. What if A performs different reads and writes after being delayed? Since we never execute past a block that interacts with any of A's variable accesses without re–enabling A, no blocks

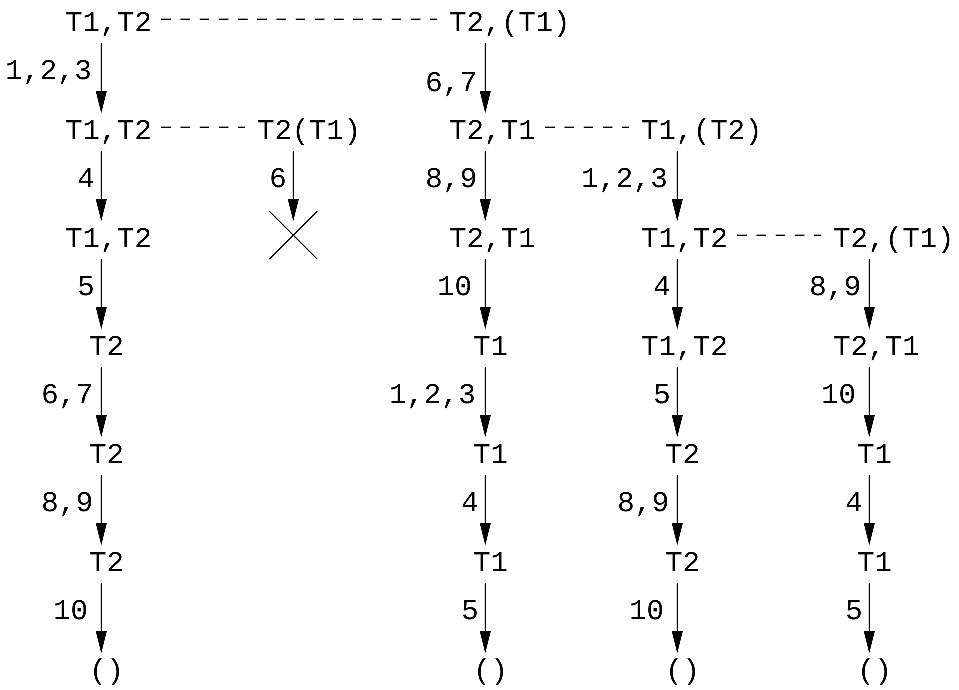

Figure 2.10: This is the tree of schedules explored by ExitBlock for the threads in Figure 2.11. Threads in parentheses are delayed. Arrows indicate the execution of the sections of code on their left. An `X` means that a path was aborted because a lock could not be obtained. (This figure is a duplicate of Figure 2.8.)

```
Thread 1 =
1:   <arbitrary code>
2:   synchronized (a) { <arbitrary code>
3:       synchronized (b) { <arbitrary code> }
4:       <arbitrary code> }
5:   <arbitrary code>

Thread 2 =
6:   <arbitrary code>
7:   synchronized (a) { <arbitrary code> }
8:   <arbitrary code>
9:   synchronized (b) { <arbitrary code> }
10:  <arbitrary code>
```

Figure 2.11: Two threads, one containing nested synchronized regions. We divide `Thread 1` into atomic blocks 1,2,3 and 4,5, and `Thread 2` into atomic blocks 6,7 and 8,9,10. There is a data dependency between code sections 2 and 7, but otherwise the two threads are independent. (This figure is a duplicate of Figure 2.6.)

executed while A is delayed can affect the reads and writes it would perform upon being re–enabled.

The ExitBlock-RW algorithm's schedules for the threads in Figure 2.11, assuming that the only inter–thread data dependency is between sections 2 and 7, are shown in Figure 2.13.

2.4.1 Number of Schedules Executed by ExitBlock-RW

If no blocks interact, no thread we delay ever becomes reenabled. For each schedule, each thread runs a certain amount, gets delayed, and never wakes up. (Of course, there is at least one schedule for each thread that executes that thread until it dies.) Thus we can consider the problem of creating each schedule simply that of deciding where to cut off each thread; by arranging the thread sections end–to–end we have the schedule. The sections have the property that the section of thread i must precede that of thread j if i is started first in the program.

For a program with k threads that each obtain locks a total of n times, with absolutely no interactions between the atomic blocks, we have $n+1$ places to terminate each thread, and it does not matter where we terminate the last thread of each schedule; thus we have $(n+1)^{k-1}$ different schedules. This number will be lower if some threads cannot run until others finish, or other constraints are present, and higher if interactions between blocks exist (potentially as high as ExitBlock's formula if every block interacts with every other, which fortunately is very unlikely).

This best–case result, polynomial in the number of locks per thread and exponential in the number of threads, is much better than the growth of ExitBlock which is exponential in the number of locks per thread. The number of threads in a program is typically not very high, even for large programs, while the code each thread executes can grow substantially. Thus in the best case the ExitBlock-RW algorithm achieves polynomial growth.

The important question is, how often is the best case achieved? The number of interactions between threads in a program is usually kept to a minimum for ease of programming. So it appears likely that the typical number of paths that ExitBlock-RW needs to explore is closer to the best case result than the worst case result.

```
BEGIN:
 run program normally until 2nd thread is started, then:
     delayed = {}
     ...
LOOP:
 if (enabled is empty)
     ...
     pop (RetPt, saved_enabled, saved_delayed, wait,
         no_notify) off of stack
     change execution state back to RetPt
     delayed = saved_delayed
     reads = writes = {}
     ...
 ...
 run curThread until one of the following events occurs:
     if (curThread performs a read)
         record it in the reads set
     if (curThread performs a write)
         record it in the writes set
     ...
     if (lock_exit)
         save_enabled = old_enabled minus curThread
         save_delayed = delayed plus (curThread, reads, writes)
         push on stack (RetPt, save_enabled, save_delayed, wait, {})
         foreach (thread, r, w) in delayed
             if ((r,w) intersects (reads,writes))
                 move thread from delayed set to enabled set
         move all threads in lock's blocked set to enabled set
         goto LOOP
DONE: // testing is complete!
```

Figure 2.12: Changes to the ExitBlock pseudocode of Figures 2.7 and 2.9 for the ExitBlock-RW algorithm. This algorithm records the reads and writes performed during atomic blocks and only interleaves two blocks if their read and write sets intersect.

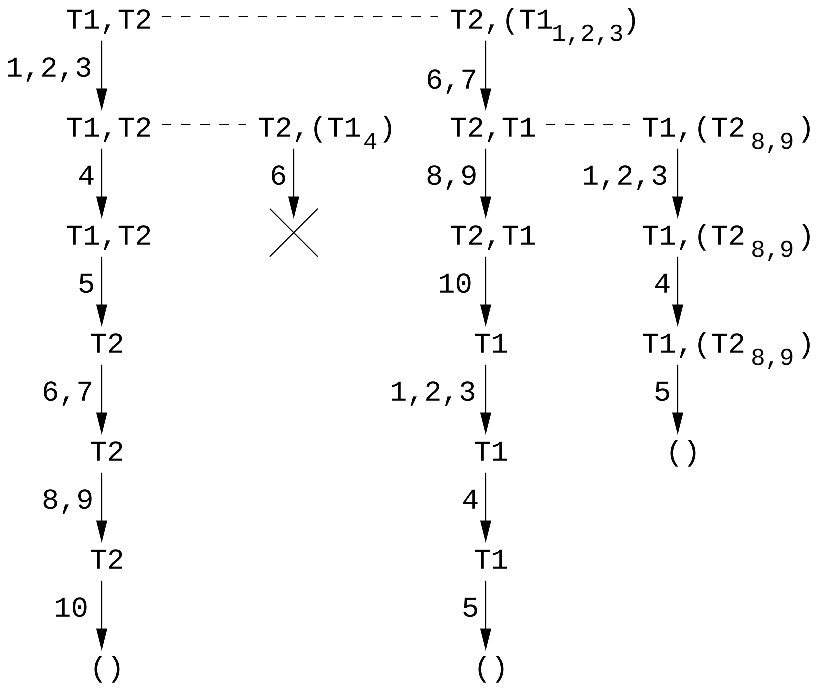

Figure 2.13: Tree of schedules explored by ExitBlock-RW for the threads in Figure 2.6 on page 34, with the only inter–thread data dependency between code sections 2 and 7. Threads in parentheses are delayed, with subscripts indicating the code sections over which read/write intersections should be performed. Compared to the tree for ExitBlock shown in Figure 2.10, this tree does not bother finishing the schedule 6,7,1,2,3,4,5,. . . and does not execute at all the final schedule of the other tree (6,7,1,2,3,8,9,10,4,5).

2.4.2 Correctness Proof for ExitBlock-RW

Theorem 4 *Consider a program P that meets the testing criteria defined in Section 2.1. Suppose that for a given input and when executed on a single processor P produces an assertion violation. Let $T = \{t_1, \ldots, t_n\}$ be the set of threads that P started during its execution, and let $S = s_1 s_2 \ldots s_m$ be the schedule of these threads that led to the assertion violation. A schedule S'' exists that is produced by the ExitBlock-RW algorithm in which the same assertion is violated.*

Proof: We start by applying the transformations listed in the proof of Theorem 3 to S, resulting in an exit–block schedule S' that leads to the same assertion violation as S. Now we apply the following transformation to each segment s'_i of S':

If s'_i has a segment s'_{i-1} to its left (i.e., $i > 0$), and s'_{i-1} is not a segment of the same thread as s'_i, and s'_i has no data dependencies with s'_{i-1}, swap the order of the two segments. Move each segment in this way as far as possible to the left.

This transformation shifts segments to the left over all intervening independent segments. The resulting schedule S'' contains no segments that can be shifted to the left in this manner. Furthermore, S'' contains the same assertion violation as S, since only the order of independent segments has changed from S'.

ExitBlock-RW produces, for each schedule produced by ExitBlock, a prefix (not necessarily proper, but necessarily not empty) of that schedule. The prefix cannot be empty because ExitBlock-RW starts with the same set of enabled threads as ExitBlock and produces a separate top-level branch for each one, just like ExitBlock. Prefixes that are ended short of the full schedule contain at the end at least one delayed thread that was not re–enabled.

Assume that S'' is not a schedule produced by ExitBlock-RW. Since S'' is produced by ExitBlock (the transformations from S' to S'' do not change the fact that the schedule is an exit–block schedule, since the composition of each segment is unchanged), ExitBlock-RW must produce a prefix of S''. This means that at least the final atomic block of S'' was independent of the stored reads and writes of all delayed threads in the delayed set that was present when the prefix ended (otherwise at least one delayed thread would have been re–enabled and S'' would be longer than it is). But this is a contradiction, since then S'' could have further transformations applied to it. Thus we must conclude that S'' is a schedule produced by ExitBlock-RW. $\square$

2.5 Detecting Deadlocks

A deadlock is a cycle of resource dependencies that leads to a state in which all threads are blocked from execution. Two kinds of cycles are possible in Java programs; if the cycle is not one of locks, then it must involve some or all threads in a `wait` state and the rest blocked on locks. We will refer to deadlocks consisting solely of threads blocked on locks as *lock–cycle deadlocks*, which will be discussed in the next section, and those that contain waiting threads as *condition deadlocks*, which will be discussed in Section 2.5.2.

2.5.1 Lock–Cycle Deadlocks

The ExitBlock algorithm rarely executes schedules that result in lock–cycle deadlocks. (When it does it simply aborts the current branch in the tree of schedules.) Consider

```
Thread 1 =
1:   <arbitrary code>
2:   synchronized (a) { <arbitrary code>
3:       synchronized (b) { <arbitrary code> }
4:       <arbitrary code> }
5:   <arbitrary code>

Thread 2 =
6:   <arbitrary code>
7:   synchronized (b) { <arbitrary code>
8:       synchronized (a) { <arbitrary code> }
9:       <arbitrary code> }
10:  <arbitrary code>
```

Figure 2.14: Two threads with the potential to deadlock. If the code sections are executed in any order where 2 precedes 8 and 7 precedes 3 (for example, 1,2,6,7) then a deadlock is reached in which `Thread 1` holds lock `a` and wants lock `b` while `Thread 2` holds lock `b` and wants lock `a`.

the threads in Figure 2.14 and the schedules that the ExitBlock algorithm produces for these threads, shown in Figure 2.15. In order for deadlock to occur, `Thread 1` needs to be holding lock `a` but not lock `b` and `Thread 2` needs to be holding lock `b` but not lock `a`. This will not happen since for ExitBlock the acquisitions of both locks in each thread are in the same atomic block. Deadlocks are only executed in rare cases involving multiply nested locks.

In order to always detect deadlocks that are present, we could change our definition of atomic block to also end blocks before acquiring a lock while another lock is held. This would cause ExitBlock to directly execute all schedules that result in lock–cycle deadlocks; however, this is undesirable since it would mean more blocks and thus many more schedules to search. Instead of trying to execute all deadlocks we execute our original, minimal number of schedules and detect deadlocks that *would occur* in an unexplored schedule.

The key observation is that a thread in a lock–cycle deadlock blocks when acquiring a *nested* lock, since it must already be holding a lock. Also, the lock that it blocks on cannot be the nested lock that another thread in the cycle is blocked on, since two threads blocked on the same lock cannot be in a lock cycle. The cycle must be from a nested lock of each thread to an already held lock of another thread. For example, when the threads of Figure 2.14 deadlock, `Thread 1` is holding its outer lock `a` and blocks on its inner lock `b` while `Thread 2` is holding its outer lock `b` and blocks on its inner lock `a`.

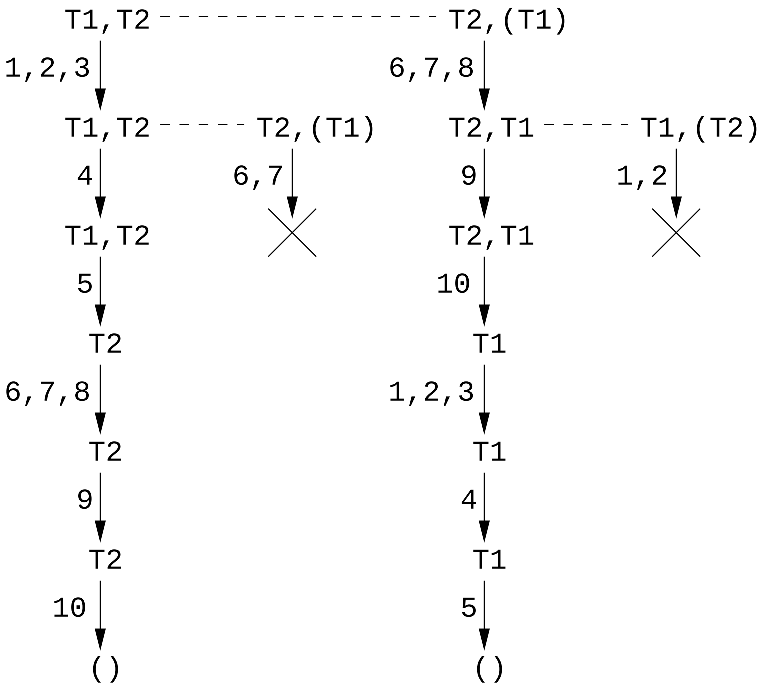

Figure 2.15: Tree of schedules explored by ExitBlock for the threads in Figure 2.14. Reverse lock chain analysis will detect two deadlocks, one at each aborted path (denoted by a large `X` in the figure). Threads in parentheses are delayed.

These observations suggest the following approach. We track not only the current locks held but also the last lock held (but no longer held) by each thread. Then, after we execute a synchronized region nested inside some other synchronized region, the last lock held will be the lock of the inner synchronized region. When a thread cannot obtain a lock, we look at the last locks held by the other threads and see what would have happened if those threads had not yet acquired their last locks. We are looking for a cycle of matching outer and inner locks; the outer locks are currently held by the threads and the inner locks are the threads' last locks held. If the current thread cannot obtain a lock l and we can follow a cycle of owner and last lock relationships back to the current thread — if l's current owner's last lock's current owner's last lock's ... ever reaches a current owner equal to the current thread — then a lock–cycle deadlock has been detected. We call this *reverse lock chain analysis*. It is straightforward to implement, and since failures to acquire locks are relatively rare it does not cost much in performance.

In Figure 2.15, the two large `X`'s indicate paths that were terminated because a lock could not be obtained. These are the points where reverse lock chain analysis occurs. At

```
Thread 1 =
    <arbitrary code>
    synchronized (a) { <arbitrary code> }
    <arbitrary code>
    synchronized (b) { <arbitrary code> }
    <arbitrary code>

Thread 2 =
    <arbitrary code>
    synchronized (b) { <arbitrary code> }
    <arbitrary code>
    synchronized (a) { <arbitrary code> }
    <arbitrary code>
```

Figure 2.16: Two threads that do not fool reverse lock chain analysis. At first glance these threads appear to contain what the analysis would report as a lock cycle; however, the analysis will never even be performed since there are no nested locks.

the first point, thread `T1` holds lock `a` and last held lock `b`. Thread `T2` holds `b` and fails in its attempt to obtain `a`. The analysis performed at this point finds the following cycle: `a`'s current owner is `T1`, whose last lock is `b`, whose current owner is `T2`, the current thread. At the second point a similar analysis discovers the same deadlock in a different schedule.

Reverse lock chain analysis does not detect false deadlocks. For example, although the threads in Figure 2.16 contain a cycle of last lock held and currently owned relationships, since there are no nested locks in either thread there will be no failures to obtain locks and thus no analysis will be performed.

We can improve performance by using reverse lock chain analysis with ExitBlock-RW rather than with ExitBlock. This seems to work well in practice. For all of the example programs in Chapter 4 that contain lock–cycle deadlocks, ExitBlock-RW plus analysis finds them.

However, this optimization comes at a cost. ExitBlock-RW plus reverse lock chain analysis does not always find deadlocks that are present. Figure 2.17 shows a counterexample. ExitBlock plus analysis finds the deadlock in this program, while ExitBlock-RW does not. The tree of execution paths for ExitBlock on the program is shown in Figure 2.18. The potential deadlock is detected in two different places. Figure 2.19 shows the tree of schedules for ExitBlock-RW on the program (none of the regions of code of the two threads interact with the other thread at all). The pruning performed by ExitBlock-RW has completely removed the branches of the tree that detect the deadlock.

```
Thread 1 =
    <arbitrary code>
    synchronized (a) { <arbitrary code>
        synchronized (b) { <arbitrary code> }
        <arbitrary code> }
    <arbitrary code>

Thread 2 =
    <arbitrary code>
    synchronized (a) { <arbitrary code> }
    <arbitrary code>
    synchronized (b) { <arbitrary code>
        synchronized (a) { <arbitrary code> }
        <arbitrary code> }
    <arbitrary code>
```

Figure 2.17: Two threads whose potential deadlock will be caught by reverse lock chain analysis in ExitBlock (see Figure 2.18) but not in ExitBlock-RW (see Figure 2.19). None of the regions of code interact.

An idea we have not yet explored is to have ExitBlock-RW consider a lock enter to be a write to the lock object; this would certainly prevent the pruning of the deadlock situations in this example, but we have not proved it would always do so. We do prove in Section 2.5.3 that ExitBlock plus reverse lock chain analysis guarantees to find a deadlock if one exists. In the next section we discuss detecting condition deadlocks.

2.5.2 Condition Deadlocks

Condition deadlocks involve a deadlocked state in which some of the live threads are waiting and the rest are blocked on locks. The tester can detect condition deadlocks by simply checking to see if there are threads waiting or blocked on locks whenever it runs out of enabled threads to run. We call the combination of this checking with ExitBlock and reverse lock chain analysis the ExitBlock-DD algorithm. The same combination but with ExitBlock-RW we call ExitBlock-RWDD.

ExitBlock-RWDD cannot use the same condition deadlock check as ExitBlock-DD because of its delayed thread set. We only delay threads to prune paths, so if we end a path with some waiting threads but also some delayed threads, we have not necessarily found a deadlock since nothing is preventing the delayed threads from executing and waking up the waiting threads. As an example, consider the programs in Figure 2.20 and Figure 2.21. Fig-

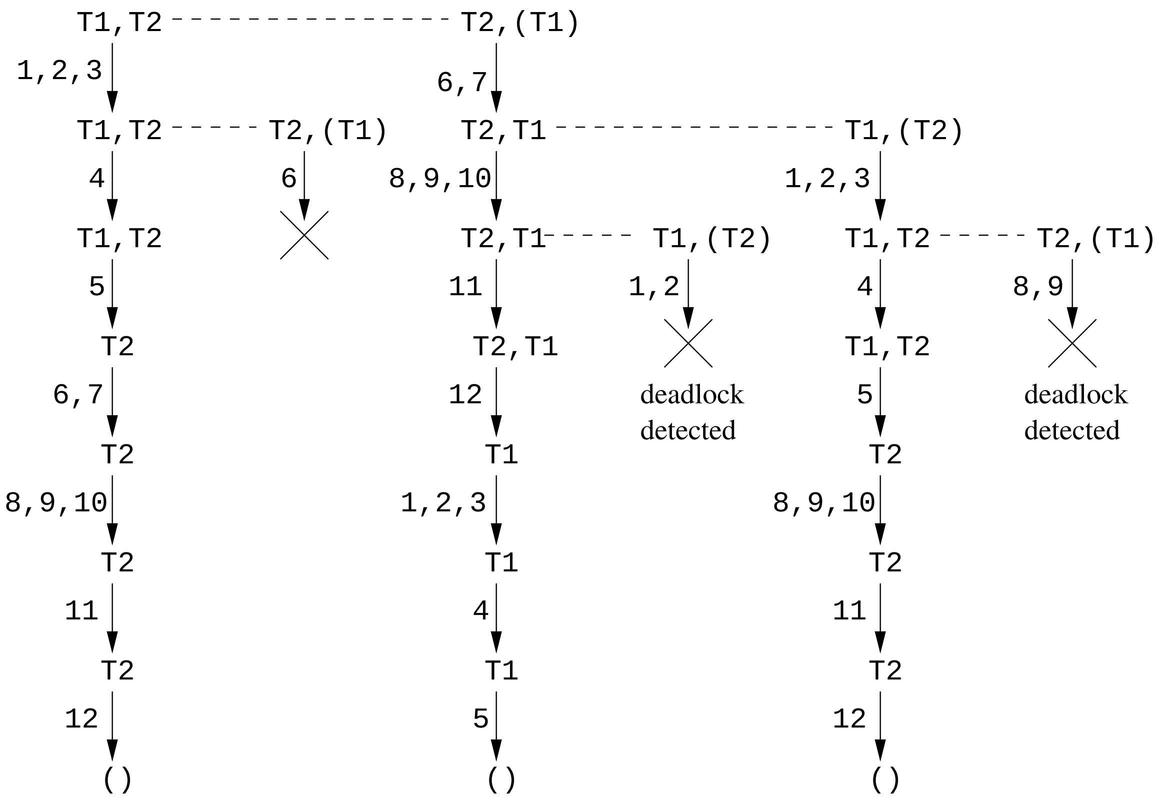

Figure 2.18: Tree of schedules explored by ExitBlock for the threads in Figure 2.17. Threads in parentheses are delayed.

ure 2.20 contains a program that will condition deadlock if `Thread 2` runs before `Thread 1`. Figure 2.21 contains a program that will never condition deadlock. If we report a condition deadlock when a path ends with any threads waiting even if threads are delayed, then we will correctly report the condition deadlock in Figure 2.20 but we will incorrectly report a condition deadlock for Figure 2.21. Figure 2.22 shows the trees for the two programs.

Thus, to avoid reporting false condition deadlocks in ExitBlock-RWDD, we must not report condition deadlocks when there are delayed threads. We could attempt to execute the delayed threads to find out if there really is a condition deadlock; however, there is no way to know how long they might execute. We have not fully investigated this idea. ExitBlock-RWDD does successfully detect condition deadlocks in the example programs in Chapter 4, and we can prove that ExitBlock-DD will find a deadlock if one exists. Because of this, the implementation of our tester has two modes: the default uses ExitBlock-RWDD for efficiency, while the second mode uses ExitBlock-DD in order to guarantee to find deadlocks.

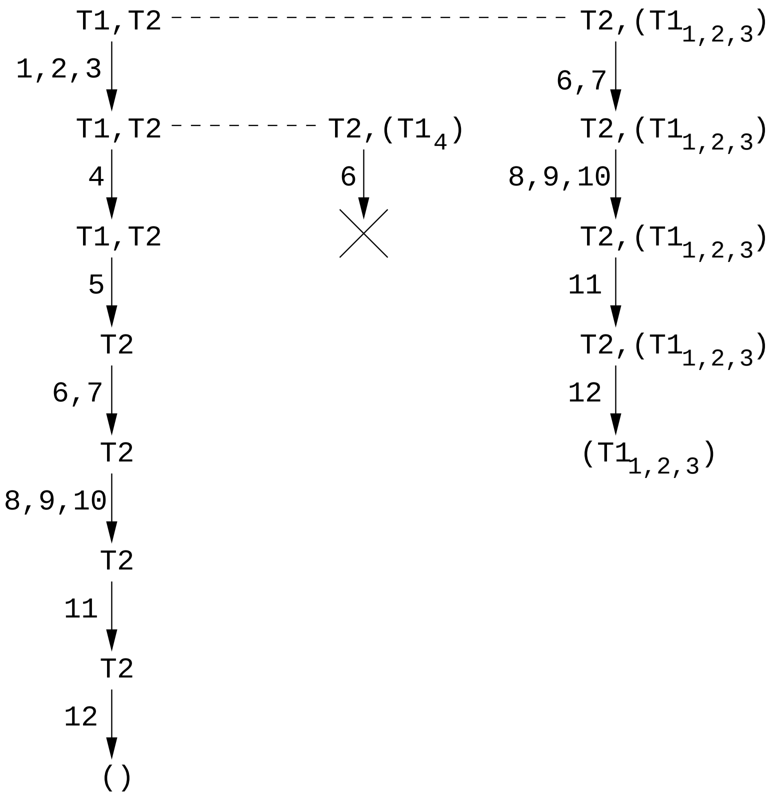

Figure 2.19: Tree of schedules explored by ExitBlock-RW for the threads in Figure 2.17. Threads in parentheses are delayed, with subscripts indicating the code sections over which read/write intersections should be performed.

```
Thread 1 =
1:   <arbitrary code>
2:   synchronized (a) { a.wait();
3:       a.notify(); }
4:   <arbitrary code>

Thread 2 =
5:   <arbitrary code>
6:   synchronized (a) { a.notify();
7:       a.wait(); }
8:   <arbitrary code>
```

Figure 2.20: If `Thread 2` executes before `Thread 1` there will be a condition deadlock — both threads will be waiting and there will be no one to wake them up.

```
Thread 1 =
1:   <arbitrary code>
2:   synchronized (a) { a.notify();
3:       a.wait();
4:       a.notify(); }
5:   <arbitrary code>

Thread 2 =
6:   <arbitrary code>
7:   synchronized (a) { a.notify();
8:       a.wait();
9:       a.notify(); }
10:  <arbitrary code>
```

Figure 2.21: Two threads that have no potential condition deadlocks. However, ExitBlock-RW's pruning (see Figure 2.22) leads it to end a path with `Thread 1` waiting; it cannot tell the difference between this and the real condition deadlock present in the threads in Figure 2.20.

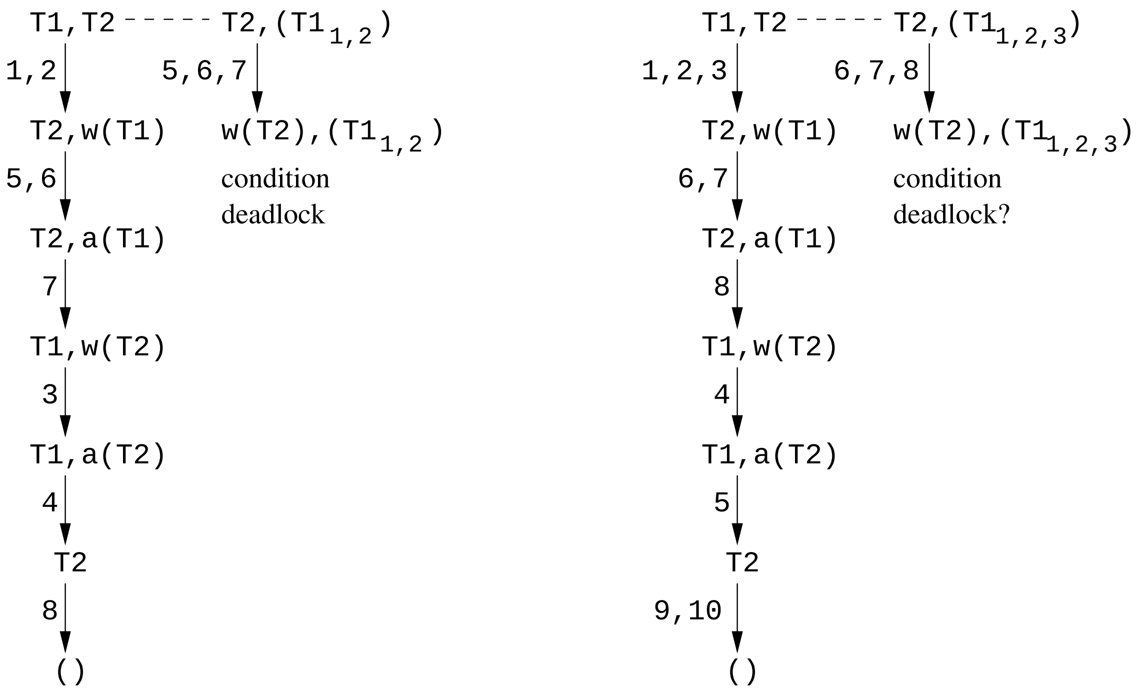

Figure 2.22: The tree on the left is the tree of schedules explored by ExitBlock-RW for the threads in Figure 2.20. The tree on the right is for the threads in Figure 2.21. Threads in parentheses preceded by a `w` are in the wait set, preceded by an `a` are in the blocked set for lock `a`, and in plain parentheses are delayed (with subscripts indicating the code sections over which read/write intersections should be performed).

```
Thread 1 =
<arbitrary code>
synchronized (a) { <arbitrary code>
    synchronized (b) { a.wait();
        <arbitrary code> }
    <arbitrary code> }
<arbitrary code>

Thread 2 =
<arbitrary code>
synchronized (b) { <arbitrary code>
    synchronized (a) { a.wait();
        <arbitrary code> }
    <arbitrary code> }
<arbitrary code>
```

Figure 2.23: These two threads contain both a lock–cycle deadlock and a condition deadlock. The tester will only detect the condition deadlock, however.

2.5.3 Correctness Proof for ExitBlock-DD

ExitBlock-DD guarantees to find a deadlock if one exists, but no guarantees are made on what type of deadlock will be found if both types exist. For example, when ExitBlock-DD is run on the program in Figure 2.23 it detects the condition deadlock but not the lock–cycle deadlock.

Theorem 5 *Consider a program P that meets the testing criteria defined in Section 2.1.*
(a) Suppose that P enters a deadlock for a given input when executed on a single processor. Let $S = s_1 s_2 \ldots s_m$ be the schedule of the set of threads $T = \{t_1, \ldots, t_n\}$ that led to the deadlock. Then a schedule is produced by the ExitBlock-DD algorithm in which a deadlock is detected (though not necessarily the same one).
(b) Furthermore, if there are no schedules of P that result in a deadlock, the ExitBlock-DD algorithm will not detect any deadlocks.

Proof of (a): First we prove that if a deadlock can occur, ExitBlock-DD will detect one. We apply the same transformations as in the proof of Theorem 3 to S to get S'. We consider the final segment of each deadlocked thread to end in "thread death" to satisfy that theorem's assumptions. This means that threads can be holding locks when they "die", but Theorem 3 does not require that they do not. Schedule S' is an exit–block schedule except for the final segments of the deadlocked threads.

Now we proceed by cases on what type of deadlock P entered. The three cases are differentiated by the number of threads waiting — all, some, or none. The first two cases are condition deadlocks while the third is a lock–cycle deadlock.

1) If all deadlocked threads are waiting, then S' is an exit–block schedule since all the deadlocked threads end in a lock exit (`wait` performs a lock exit). Thus ExitBlock-DD will execute S' and detect the deadlock.

2) If at least one deadlocked thread is waiting and at least one is blocked on a lock, we first discuss a special sub–case. The argument for the sub–case in which all but one of the deadlocked threads are waiting (the other is blocked on a lock) is simpler than the general case. If all threads but one in S' are waiting ExitBlock-DD will attempt to execute S' since it tries every exit–block schedule. All threads but the one blocked on the lock are at lock exits, and ExitBlock-DD will attempt to execute the non–waiting thread; when it fails to obtain its lock, the path will terminate and the condition deadlock will be detected.

For the general case in which at least one of the deadlocked threads is waiting, but not all (the rest are blocked on locks), let S'' be the schedule that is identical to S' except that the threads blocked on locks have not executed as far as they have in S' — each has just finished its respective previous lock exit. This is possible because backing them up in this way cannot affect interactions between them: if one of them modifies a shared variable, no other thread can view the change until the modifying thread releases the lock protecting the variable. Since we have not crossed any lock exits in backing up, there can be no communication between the backed–up threads. Now ExitBlock-DD will try all exit–block schedules starting with S'', a partial exit–block schedule. We know that one order of execution of the backed–up threads leads to a condition deadlock, so ExitBlock-DD must detect that deadlock since it tries all orders of threads from S''.

3) If all deadlocked threads are blocked on locks, we have a lock–cycle deadlock and so there must be a cycle of lock dependencies. Each thread t_i in the cycle must be holding a lock that another of the threads wants; call that lock h_i. Call the lock that t_i wants w_i. Thus every w_i matches up with an h_j, $i \neq j$, and vice versa. There must be a cycle in the relationship between w and h. Since thread t_i is holding h_i when it attempts to acquire w_i, and since locks in Java must be properly nested, the synchronized region of w_i must be completely contained inside of the synchronized region of h_i.

Let S'' be the schedule identical to S' except that each thread t_i is backed up to the lock exit prior to its acquisition of h_i. We can do this without affecting program behavior by the same argument as in case 2. ExitBlock-DD will execute the partial exit–block schedule S'' and will attempt to execute all thread orderings from S''. We want to execute them forward beyond where they deadlocked to get them into a configuration where every thread t_i has just released w_i, except for one thread t_j who blocks acquiring w_j. At this point reverse lock chain analysis will detect the lock–cycle deadlock. However, many things could prevent the threads from executing forward to reach that configuration. We need to show that all of those obstacles produce other deadlocks that will be detected.

Each thread t_i will execute forward, obtaining and then releasing w_i. This will happen in S'' since in S' it was trying to obtain w_i and thus eventually releasing it; this behavior cannot be different in S''. The behavior cannot change since if some other thread t_j modifies a shared variable to communicate with t_i, t_j has to have held the lock for that variable before it obtained h_j, and therefore cannot exit that lock to allow t_i to read the shared variable until after it exits h_j. Thus each t_i will attempt to execute forward until the exit of w_i. It may not reach the exit, however. We split this case into two sub–cases:

A. If each thread t_i up to one last thread t_j $(i \neq j)$ is successful in reaching the lock exit for w_i, then t_j will successfully obtain h_j (since the thread that wanted it has released it) but not w_j (since the other threads still hold their h_i's). Note that this can only happen if the threads are executed in an order such that if $w_i = h_j$ then t_i executes before t_j. At this point reverse lock chain analysis will see that t_j cannot obtain w_j, whose current owner must be one of the t_i, say t_{j2} $(j2 \neq j)$, whose last lock held must be w_{j2} (because every t_i but t_j just executed the lock exit of w_i), whose current owner must be yet another of the t_i, say t_{j3} $(j3 \neq j2 \neq j)$, etc. Because the $w_i \leftrightarrow h_j$ relationship is a cycle no matter what w_i we start with, we will end up at the one who started it all, h_j, whose owner, t_j, is the current thread. Thus ExitBlock-DD will report a lock–cycle deadlock.

B. If more than one thread does not reach its destination, then each such thread t_i either performs a `wait` on some object whose lock is not equal to w_i or blocks on a lock. This prevents the next thread in the cycle from executing, which will prevent the next thread, and so on. (Only a `wait` on the object whose lock is w_i will not do so, but it will involve a lock exit of w_i.) Only the threads in the cycle that have already been executed will not be stuck. If those threads can execute to completion and enable one of the remaining threads to execute (by either performing a `notify` or releasing a lock) then we have a condition deadlock in a different schedule where the free threads execute to completion first and the rest of the threads are now stuck with no threads to free them on their way to the lock exit for w_i. If the free threads cannot free the other threads then we have a condition deadlock also, in this schedule. Thus if more than one thread t_i cannot execute to the lock exit of w_i then a deadlock — not the lock–cycle one, but some condition deadlock — will be detected.

This proves the first part of the theorem.

Proof of (b): Now we need to show that if ExitBlock-DD detects a deadlock, that deadlock can occur (i.e., we produce no false positive results). If reverse lock chain analysis finds a lock cycle, and we back up every thread in the cycle (except for the current thread) to before the lock enter for its last lock held, the threads will still be holding the same locks as in the deadlock because Java locks are properly nested. Now every thread is about to execute a lock enter on a lock that another thread holds — the system can clearly reach a deadlocked state. Thus, if ExitBlock-DD's reverse lock chain analysis ever claims a lock–cycle deadlock could occur, there exists a schedule that does deadlock. As for condition deadlocks, if a schedule ever ends with all live threads blocked on locks or waiting, then that schedule obviously represents a real condition deadlock. □

2.6 Enhancements

Two ideas mentioned in Section 2.5 have not been investigated yet. One is to have ExitBlock-RW consider a lock enter to be a write to the lock object, which will allow it to detect more lock–cycle deadlocks. The penalty in extra paths considered should not be too high since atomic blocks sharing the same lock often have data dependencies anyway.

The other idea is to have ExitBlock-RW, when it would declare a condition deadlock except for delayed threads being present, execute those delayed threads to determine if it truly is a condition deadlock. This may have performance penalties since the delayed threads could end up executing for a long time. However, if these two ideas combined provided a guarantee that ExitBlock-RW found a deadlock if one existed, they could be turned off in the more efficient, no deadlock guarantee mode. We have not yet examined whether they do indeed provide a guarantee.

Further pruning of the schedules considered by ExitBlock-RW is possible while preserving the guarantee that all assertion violations are detected. If the user knows that one of the program's methods is used in such a way that it causes no errors in the program, he or she could instruct the tester to not generate multiple schedules for atomic blocks contained in the method. For example, if a program uses some library routines that it assumes are thread safe and error free, the user could tell the tester to trust those library routines. Then every time the tester would normally end an atomic block, it would check to see if the current method was one of the trusted routines; if so, it would simply keep executing as though there was not an atomic block boundary there. One instance of this extension is implemented in our tester as the `javasafe` option described in Section 3.3, which assumes that all methods in any of the `java.*` packages are safe. As shown by the sample programs in Chapter 4, this option dramatically reduces the number of paths the tester must explore. In the future this option will become more flexible and allow the user to specify more than just the core JDK methods.

Another pruning idea is to ignore atomic blocks whose lock object is not shared. If only one thread ever accesses it then there is no reason to consider possible schedules encountered while in methods called on that object. Static analysis would be required to identify objects that are not shared, since dynamically the tester can never be sure if the next bytecode will end up copying a local object into some shared variable. Static analysis could provide other information to the tester, such as predetermining which paths are impossible (because of locks being held) so that the tester does not have to blindly find out.

Other properties of programs could be checked but are not; for example, starvation of a thread could be detected when a user–specified amount of time has elapsed since the last lock acquisition by the thread. However, this time needs to be relatively long in order for starvation to be an issue. Because the tester cannot currently test long execution sequences in a reasonable amount of time, it does not bother to check for starvation.

Chapter 3

Implementation

The systematic tester needs a good deal of control over the execution of the program being tested, called the client program. In particular, the tester needs:

- The ability to back up: undo a set of state changes in order to return to a previous state after exploring a partial schedule.
- The ability to control which threads are allowed to run and to put constraints on their order of execution.
- The ability to control which threads waiting on a condition variable are woken up when a `notify` is issued.
- The ability to record all variable reads and writes performed by the client program.
- The ability to deterministically replay all interactions between the client Java code and the rest of the system, namely input, output, and native methods. This is required to present a consistent view of the external world to the client program as the tester executes multiple schedules of the same code.

Tool interfaces for existing Java virtual machines provide features like thread control and read and write observation. However, they do not provide undo or deterministic replay. For the tester to implement these features efficiently on a standard virtual machine would be quite a challenge. We decided to incorporate checkpointing and deterministic replay directly into a virtual machine that our group was building, called Rivet.

3.1 Rivet Virtual Machine

Rivet is an extensible tool platform structured as a Java virtual machine. It is designed to support a suite of advanced debugging and analysis tools for Java programmers. The systematic tester is one of the initial tools in this suite. Other tools under development include a tracer and a bidirectional debugger based on [Boo98]. This debugger will support execution in the reverse direction of program flow, including stepping, execution to the next breakpoint, and execution until a data value changes. The tester will be integrated with the debugger so that the tester can be launched from any point while debugging a program.

Many testing, debugging, and analysis tools include similar functionality, such as access to program data and instructions during execution and control over thread scheduling and input and output events. These features are already available inside every virtual machine. Rivet exposes its internal execution environment to tools in a consistent, well–documented way. Its event system allows tools access to run–time information and control, and its built–in checkpointing facilities allow tools to undo client program operations.

Another key difference between Rivet and standard Java virtual machines is that Rivet is itself written in Java. The design of Rivet focuses on modularity, extensibility, and sophisticated tool support rather than on maximizing performance. We expect Rivet to be used to execute tools during software development, while a separate, high–performance virtual machine is used for normal, more efficient operation.

For Rivet to apply in a range of development environments, it avoids platform–specific code. Since Rivet runs on top of another Java virtual machine, it can make use of that "lower" virtual machine's platform–specific features like garbage collection and native method libraries. This not only made developing initial versions of Rivet simpler by allowing it to delegate unimplemented functionality to the lower virtual machine, but also ensures that non–standardized features of the Java environment in the developer's virtual machine will remain available when using Rivet. Rivet is designed to use a Just In Time dynamic compiler (JIT) that translates client program Java bytecodes to bytecodes that execute directly on the lower virtual machine, relying on the lower virtual machine's JIT to perform sophisticated optimizations. Initial performance results from this JIT are encouraging, but the JIT was not available in time to be used for the experiments in this thesis.

Our systematic tester makes heavy use of Rivet's checkpointing. Rivet performs incremental object checkpointing by versioning client objects. Deterministic replay of input and output is not yet implemented in the tester, but it will build on Rivet's deterministic replay abilities (which do not quite match what the tester needs, as described in Section 3.3.2). Our tester also uses Rivet's event system and its extra field mechanism, which allows a tool to add fields usable for its own data storage to client classes. For details of the implementation of Rivet, see Appendix A.

3.1.1 Performance of Rivet

Tables 3.1 and 3.2 detail Rivet's performance on six different benchmarks on two different platforms, JDK1.1.5 on Linux and JDK1.1.6 on Windows NT. On JDK1.1.6 the tests were run both with the lower virtual machine's JIT disabled and with its JIT enabled (there is no JIT for JDK1.1.5 on Linux). The `R/J` columns show the Rivet time divided by the Java time, which indicates the slowdown of Rivet. Without an underlying JIT, Rivet is nearly 200 times slower than the interpreter on which it runs. With an underlying JIT, this shrinks to 86. The slowdown for the `_201_compress` test is much larger than this. For the underlying Java interpreter, the speedup that the JIT achieves on this test is much greater than that for the other tests. This is because this test spends all of its time in one loop that contains only inlinable method calls. Most Java programs are not written like this; they are written in a more object–oriented fashion with many method calls. Thus, the geometric mean is also computed excluding this benchmark. It indicates that the slowdown for Rivet with an underlying JIT is 67.

The final column of Table 3.1 shows that the speedup the JIT achieves for Rivet is very high, with an average of over 5. Rivet's uniformly high speedup causes us to expect that as high–performance virtual machines become available and JIT's get faster, the slowdown of Rivet will decrease.

In addition, Rivet's own JIT has not been completed yet. Initial performance numbers shown in Table 3.3 indicate that Rivet's JIT will lead to a factor of two speed improvement. We are also investigating automatic translation of Rivet to C.

The specific costs due to Rivet's events, extra fields, and checkpointing features, all of which are used heavily by the tester, are analyzed in detail in the sections describing each individual feature in Appendix A.

Benchmark	NT JDKD1.1.6, no JIT			NT JDK1.1.6 with JIT			noJIT/JIT	
	Java	Rivet	R/J	Java	Rivet	R/J	Java	Rivet
_201_compress	55.1	15177.3	276	5.4	2587.4	478	10.2	5.9
_202_jess	10.7	2126.5	199	6.3	335.1	53	1.7	6.3
_205_raytrace	17.9	4456.4	249	9.6	885.8	92	1.9	5.0
_209_db	7.0	1131.9	162	3.9	185.0	47	1.8	6.1
_227_mtrt	25.8	5493.8	213	14.0	1108.2	79	1.8	5.0
JavaCUP	10.0	1261.2	126	5.9	266.6	45	1.7	4.7
geometric mean			198			86	2.4	5.5
geometric mean without _201_compress			185			67	1.9	5.2

Table 3.1: Time taken in seconds to run six different benchmarks on JDK1.1.6 on Windows NT on a Pentium II 300 MHz machine with 128MB RAM. The first five benchmarks are from the SPEC JVM Client98 Release 1.01 benchmark suite [SPEC]. They were each run with the `-s10` flag. These are not official SPEC results. The JavaCUP benchmark involved running JavaCUP [CUP] on the syntax of the Java language. The Rivet JIT was not enabled.

Benchmark	Linux JDK1.1.5		
	Java	Rivet	R/J
_201_compress	138.6	25539.8	184
_202_jess	15.7	3325.5	212
_205_raytrace	29.7	6738.8	227
_209_db	12.0	1679.9	139
_227_mtrt	34.4	8284.8	241
JavaCUP	14.2	1824.5	129
geometric mean			184
geometric mean without _201_compress			178

Table 3.2: Time taken in seconds to run six different benchmarks on JDK1.1.5 on Linux on a Pentium II 200 MHz machine with 128MB RAM. The first five benchmarks are from the SPEC JVM Client98 Release 1.01 benchmark suite [SPEC]. They were each run with the `-s10` flag. These are not official SPEC results. The JavaCUP benchmark involved running JavaCUP [CUP] on the syntax of the Java language. The Rivet JIT was not enabled.

Test	Platform	Underlying JIT?	Rivet w/o JIT	Rivet + JIT	Speedup
Straight	Linux	No	141.08	42.86	3.3
	Windows	No	93.03	28.57	3.3
	Windows	Yes	9.24	2.49	3.7
Invoke	Linux	No	148.90	56.27	2.6
	Windows	No	99.84	38.40	2.6
	Windows	Yes	10.07	6.44	1.6

Table 3.3: Time taken in seconds to run 50000 iterations of two versions of a loop. The “straight” version simply contains 207 bytecodes consisting of integer operations, `getfields`, and `putfields`. The “invoke” version adds an `invokevirtual` to the loop. The platforms are Sun’s JDK1.1.6 on Windows NT on a Pentium II 300 MHz machine with 128MB RAM and JDK1.1.5 on Linux on a Pentium II 200 MHz machine with 128MB RAM.

3.1.2 Limitations of Rivet

Rivet’s implementation in Java makes some requirements on the behavior of the native methods in client programs. Rivet handles violations of these requirements in the core JDK on a case–by–case basis.

- Native methods must not construct Java objects without calling Java constructors; otherwise the objects they construct will be incompatible with Rivet’s client object representation.
- Native methods must not block; otherwise Rivet’s preemptive thread scheduler will not operate properly.
- For deterministic replay to work, native methods cannot perform any input or output.
- Neither checkpointing nor thread switches can occur in Java methods called by native methods, since Rivet cannot checkpoint the state of the lower virtual machine in order to return to a point midway through a native call. So long as any Java methods called from native methods are short this should not pose a serious problem.
- Native methods should not perform field accesses on other than the “this” object in order for Rivet’s incremental checkpointing to correctly version objects that have changed.

- Native methods must not use direct hardcoded offsets for field access; they should use the Java Native Interface. This is a consequence of Rivet's extra field mechanism (see Section A.5).
- Native methods should not keep native state, since Rivet cannot checkpoint such state.

Only the last two limitations significantly affect Rivet's applicability. Well–constructed programs' native methods normally satisfy all the other requirements. The Java Native Interface is being promoted to solve the problem of binary compatibility across multiple versions of Java virtual machines; we expect programs to use it in the future. However, current JDK native libraries (such as `java.io` use direct field offsets which Rivet handles as a special case by re–implementing those libraries.

The final requirement, that client programs keep no state at the native level, could be removed by providing a native library that a client's native code would use to allocate memory, registering it for native–level checkpointing. For more details on Rivet, refer to Appendix A.

3.2 Eraser

We have implemented the Eraser algorithm [S+97] described in Section 2.1.1. It is used by the tester to check that client programs follow the mutual–exclusion locking discipline which is essential to the ExitBlock algorithm. In addition to being runnable by the tester, Eraser is implemented as a tool in its own right.

Our implementation of Eraser for Java is considerably simpler than the implementation of Eraser described in [S+97], which operated on arbitrary Alpha binaries. For arbitrary binaries every single memory location must be considered a variable, while in Java only field instances and array elements need to be considered variables.

To store the information about each variable that the Eraser algorithm needs, we associate a hashtable with every client object and an additional table with every client class. Since each field of an object or class is a separate variable, these tables map a field name (or array index, for array objects) to information about that variable: its state (see Figure 2.1 on page 28), the thread that last accessed it, and its candidate set of locks. The candidate set of locks for a variable v, denoted $C(v)$ in [S+97], is the set of locks that was held by every thread that has accessed v at the time of the access.

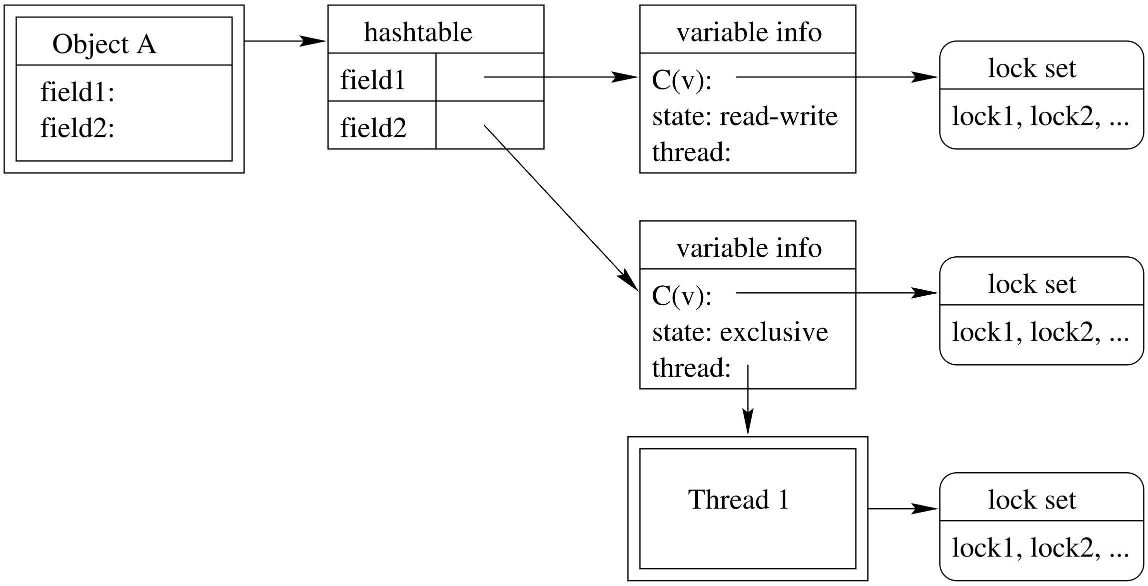

Figure 3.1: Eraser associates a hashtable with each object (and class) that holds information on its fields. For each field the table stores its state (see Figure 2.1 on page 28), the last thread that accessed it, and its candidate lock set $C(v)$. Eraser also associates a lock set with each thread.

Eraser uses Rivet's extra field mechanism (described in Section A.5) to add a field to `java.lang.Object` which holds the table for the instance fields of each instrumented object. For low–level implementation reasons, it turned out to be inefficient to use the extra field mechanism for static fields. Instead Eraser stores the tables for static fields in a hashtable keyed on client classes.

For each thread in the client program, Eraser keeps track of the collection of locks that the thread currently owns. Eraser adds a field to `java.lang.Thread` to hold this lock set. Figure 3.1 illustrates the data that Eraser stores for each object and thread.

Eraser operates by allowing the program to execute normally. On every lock enter and exit, Eraser updates the lock set of the current thread. On every field and array element read and write, Eraser updates the information associated with the accessed variable according to the rules of the algorithm. If Eraser detects a locking discipline violation it prints an error message indicating which variables and threads are involved in the violation.

Using Rivet's tool interface, implementing the Eraser algorithm was relatively simple. The main complexities came from making it checkpointable for use with the tester (see Section A.6.2).

3.3 Tester

Our tester implements the ExitBlock-RWDD and ExitBlock-DD algorithms described in Chapter 2. It has several parameters which are passed to it as command–line arguments:

eraser — Runs Eraser in parallel with the tester.

stop — This parameter causes the tester to halt execution at the first assertion failure or deadlock detected. Otherwise the tester prints a warning message and continues execution.

deadlock — Normally the tester executes ExitBlock-RWDD (path pruning plus deadlock detection). This parameter causes the tester to execute ExitBlock-DD, turning off path pruning in order to guarantee deadlock detection.

javasafe — As mentioned in Section 2.6, this causes the tester to assume that methods in classes of any of the `java.*` packages are correct. The tester ignores synchronized regions in such methods, considering their scheduling to have no impact on the program's behavior. In the future more flexible options will be available to allow the user to further aid the tester in reducing the number of schedules considered.

terminateAfter <n> — Not yet implemented. As mentioned in Section 2.1.3, this option will cause the tester to terminate all threads after **n** bytecodes, allowing use of the tester on client programs containing non–terminating threads whose source code is not available.

In order for the **stop** option to work, the tester must know when an assertion is violated. If the client source code is available and modifiable, its assertions can be changed into downcalls. Rivet's downcall mechanism is described in Section A.4. Using downcalls, the client's assertions can notify the tester when they fail. Using downcall assertions is also preferable to simply printing something out since such a printout involves synchronized regions that increase the number of schedules to consider.

3.3.1 Implementation

The tester follows the pseudocode given in Chapter 2. It uses Rivet's incremental checkpointing to undo. We considered undoing by re–executing the program from the beginning up to the point of interest using deterministic replay. However, re–execution is not efficient enough for the large number of times the tester will need to back up. When the tester is testing a section of a large program, re–executing even once may take a long time if the section occurs late in the program.

The tester associates with every object an integer identifier that it uses for quickly intersecting sets of variables, as described later. For every thread it records the last lock held by the thread — the lock object that the thread last owned, which is used in deadlock detection (see Section 2.5) — and the atomic blocks executed by the thread, which are used for debugging and replay purposes. The tester uses Rivet's extra field mechanism (described in Section A.5) to store all of this information, adding fields to `java.lang.Object` and `java.lang.Thread`.

The tester stores the information needed to execute the client program from the end of one of its synchronized regions to the end of the next (one atomic block) in an instance of the State class. This information includes: the enabled thread set; the delayed thread set with, for each delayed thread, the sets of variables read and written before it was delayed; the checkpoint identifier of the start of this atomic block; the set of threads blocked on locks; the set of threads to disable at the start of the atomic block to prevent them from being notified; and a count of threads that are waiting (to detect condition deadlocks). Although Rivet can tell the tester what threads are blocked on locks at any given time, the tester also needs to know what threads *will* block on locks if they are executed. Remember that the tester aborts paths that end with a blocked thread and returns to the previous checkpoint; thus those threads are not blocked according to Rivet and the tester must remember that they will block. (There can be threads that are truly blocked, for example, a thread just woken up from a `wait`.) One instance of the State class is stored on a stack for each branch of the tree of schedules that is yet to be explored.

The tester lets the client program execute normally until a second thread is created. The tester then begins its main loop by saving a copy of the State (hereafter referred to as "the old State") and telling Rivet to disable all the threads in its enabled set, delayed set, blocked sets, and no_notify lists — all threads, in fact, except those that are waiting and not on the no_notify list. The tester disables all these threads in order to prevent preemption and other unwanted thread switches by Rivet's thread scheduler, and also to prevent threads from being notified in branches created by `notify` operations. The no_notify list is cleared at this point since its previous value is no longer needed, as explained below.

The tester then chooses a thread from the enabled set, enables it, and executes it. If the thread blocks on a lock, after performing reverse lock chain analysis to check for a deadlock (see Section 2.5.1), the tester adds the thread to the old State's set of blocked threads and aborts the current path by returning to the old State's checkpoint and trying another thread from the enabled set. If the enabled set becomes empty, a condition deadlock is checked for (a condition deadlock exists if there are no delayed threads but there are some waiting threads or threads blocked on locks, as explained in Section 2.5.2) and the next State is popped off the stack. The tester continues its loop with the new State, disabling threads and cloning the State. When the stack becomes empty the tester is done.

If the running thread exits a lock, the tester re–enables — moves from the blocked set to the enabled set — all threads blocked on that lock (both those stored in the tester's blocked set and those really blocked), and updates the last lock held by the running thread. Then, if there are other enabled threads, the current atomic block is ended. To end the block, the tester first moves the current thread from the old State's enabled set to its delayed set, storing the reads and writes it just made in its delayed set entry. Then it moves from the old delayed set to the old enabled set all delayed entries whose reads and writes intersect with the ones the current thread just performed. How reads and writes are stored and intersected is discussed below. Finally the tester pushes the old State on the stack and continues execution of the current thread.

If the running thread performs a `wait`, the tester increments the number of waiters and removes the thread from the enabled set; it does not actually keep a set of waiting threads itself since Rivet will tell it whenever a thread is woken via a `notify`. It needs the number of waiters to detect condition deadlocks, though. For a `notify` the tester first decrements the number of waiters by the number of threads woken up, then disables all the newly woken threads, and finally, if any threads still waiting on the notify object are not disabled, clones the old State and pushes the clone on the stack. The tester cleared the no_notify set at the beginning of this atomic block, so it checks to see if threads were in the set by seeing if they are now disabled. Waiting threads are only disabled if they are in the no_notify set. This way the tester does not need to remove the originally waiting threads from the no_notify set on each `notify` operation, as Section 2.3.1 described.

The tester does not currently handle the thread operations `suspend` and `resume`. It should not prove difficult to add them, as described at the end of Section 2.3.1.

To store the reads and writes performed during an atomic block, the tester uses two instances of the VarSet class, one for reads and one for writes. Such a pair is stored with every delayed thread. Variables are represented as triples of `int`s: a field identifier, an object instance identifier (the instance whose field this is), and an offset (used for arrays only). For non–arrays, the offset is -1. For static fields, the instance is -1, while for instance fields it is the object identifier that is stored with each object. The field identifier is used for a fast intersection test (an empty intersection is the common case). A global field counter is incremented every time the tester sees a new field. The global counter begins at 1, reserving 0 for arrays. The reason the tester has its own counter is to keep field identifiers low and evenly spread from 0 to 31. On every variable access the tester adds the triple for that variable to the appropriate VarSet. VarSet stores, in addition to a set of triples identifying the variables, a single `int` called `fields` whose n^{th} bit is on if and only if there is a variable in the set with a field identifier *fid* such that $\mathit{fid}\ \mathit{mod}\ 32 = n$. The fast intersection test between VarSets `vs1` and `vs2` is then $((\texttt{vs1.fields}\ \&\ \texttt{vs2.fields}) \neq 0)$. Only if this test passes do we need to check each element of the sets against each other.

The tester, like Eraser, was relatively simple to implement on top of Rivet. Rivet's extra fields, checkpointing, and deterministic replay do the hard work for the tester. The most difficult part of implementing the tester was, as for Eraser, handling extra fields that must be checkpointed. As explained in Section A.6.2, the current methods for making a tool's extra fields checkpointable are complicated and errors in the use of checkpointing can be difficult to debug.

3.3.2 Deterministic Replay

The tester needs deterministic replay of interactions between Java code and the rest of the system. This is a different type of deterministic replay than that provided by Rivet (which is described in Section A.7) — the tester wants to go back in time and execute forward on a brand–new path in which different threads or methods than before may request the same input or output or call the same native methods as the previous path. We want these events to behave as though another path has not already executed. We cannot blindly replay the events, though, since on the new path a thread could write to a file followed by a read to the same file that happened before. We should re–execute the read to get the recently written data instead of simply supplying the incorrect logged data.

We have not yet implemented the tester's deterministic replay. However, it will require Rivet to change its replay to have all input and output pass through an intermediary that the tester can register with (this intermediary idea is also discussed in Section A.7). The tester would simply receive a copy of input or output events when not replaying, and would be able to supply data to replay events instead of re–executing them. The tester itself will need to store all input and output operations and decide when to replay from its log and when to issue the actual operation. Native methods may be a challenge; to replay them we should restrict them to only modifying fields of "this" and then log their modifications, or else assume that they are functional with respect to their arguments and the Java state of the system so that we can simply re–execute them. For now we take this latter approach. We also assume, until we have implemented deterministic replay, that input and output operations are repeatable (for example, all reads are from files whose contents do not change).

These assumptions are additional requirements for client programs on top of the testing criteria of Section 2.1. We must also add Rivet's requirements on native methods (Section 3.1.2 summarizes them).

Once Rivet supports graphics, asynchronous AWT events will have to be logged and replayed. This could be challenging.

3.3.3 Performance

Profiles of testing runs indicate that approximately one–third of the tester's execution time is spent on Rivet's checkpointing. Table 3.4 shows the percentage of time spent making checkpoints and returning to checkpoints for several testing runs. Improving the performance of Rivet's checkpointing will improve the performance of the tester dramatically. Section A.6.3 discusses the performance of Rivet's checkpointing.

Memory management is critical because the tester runs for so long. Table 3.5 lists the ratio of number of checkpoints created versus number returned to for various testing runs. As can be seen, at least 7 checkpoints are created for every one returned to. The number gets much higher when there are many synchronized regions that do not interact with any other regions, as in the Performance tests. The Performance program is presented in Section 4.1; it takes two parameters, the first specifying how many threads to create and

Program	Tester Parameters	Paths	Creating	Returning To	Total
Performance 5 5	javasafe	2801	26%	8%	34%
BufferIf		*7958	17%	11%	28%
Deadlock3	deadlock	*1250	19%	9%	28%
Deadlock3	deadlock	*5280	23%	10%	33%
Deadlock3	deadlock	*19980	22%	11%	33%
Deadlock3	deadlock	*25007	22%	11%	33%

Table 3.4: Percentage of time spent creating checkpoints and returning to checkpoints for several testing runs. A * indicates that the program was not tested in its entirety. These numbers are for JDK1.1.6 on Windows NT on a Pentium 300MHz machine with 128MB RAM. They were measured with JProbe Profiler [JProbe]. The programs tested are described in detail in Chapter 4.

the second how many locks each thread should acquire. The threads do nothing but acquire that many locks each and so do not have data dependencies with each other.

In programs with large numbers of synchronized regions, a strategy of not taking checkpoints at program points that are very close together and using deterministic replay to execute to the desired back–up point may be more efficient than taking a checkpoint after every synchronized region. Another optimization is to not take checkpoints at the beginning of atomic blocks when there is only one live thread, since such a checkpoint would never be used.

Chapter 4 gives execution times and the number of schedules executed in testing a number of example programs.

3.3.4 Future Work

Deterministic replay (described in Section 3.3.2) is unimplemented, as well as the thread operations `suspend` and `resume`; these are all future work items.

To reduce the number of checkpoints taken, the ideas of the previous section should be explored. These ideas are to not take checkpoints when there is only one live thread, and to take fewer checkpoints when there are many short atomic blocks, using deterministic replay to undo to precise locations.

The tester's algorithm cannot handle asynchronous input, such as AWT events.

Program	Tester Parameters	Paths	Created	Returned To	Ratio
SplitSync		18	131	17	7.7
NoEraser		16	129	15	8.6
Deadlock	deadlock	841	6564	848	7.7
Deadlock3		79	665	84	7.9
DeadlockWait	deadlock	19	183	24	7.6
BufferIf		22958	153286	22957	6.7
BufferNotify		29350	208650	29349	7.1
Performance 2 1		10	71	9	7.9
Performance 2 2		11	89	10	8.9
Performance 2 3		12	109	11	9.9
Performance 2 100		109	11555	108	107.0
Performance 2 250		259	66305	258	257.0
Performance 2 500		509	257555	508	507.0
Performance 2 1000		1009	1015055	1008	1007.0
Performance 3 1		67	487	66	7.4
Performance 3 2		85	691	84	8.2
Performance 3 3		105	947	104	9.1
Performance 3 50		3301	182179	3300	55.2
Performance 3 100		11551	1214029	11550	105.1

Table 3.5: Number of checkpoints created and returned to and the ratio of checkpoints made to checkpoints returned to for various testing runs of the programs presented in Chapter 4. These numbers were recorded on JDK1.1.6 on Windows NT. The number of checkpoints returned to is equal to the number of paths tested, minus one, plus the number of paths aborted because of un–acquirable locks. The Performance program (presented in Section 4.1) takes two parameters, the first indicating how many threads to create and the second how many locks each thread should acquire in its lifetime.

Asynchronous input could be handled by having an always–enabled thread providing it. This thread would be scheduled everywhere possible and would never be delayed.

The ability to replay a path is unimplemented. It could make use of information used in deterministic replay discussed in Section 3.3.2, but it would have to store the logs in files for use in a separate tester invocation. Since schedules are typically not executed without interruption from the start to the end, but rather in pieces as the tester does its depth–first search, replaying schedules would involve piecing together logged information.

Running the tester on sections of programs is also unimplemented. Perhaps down-calls could be inserted in the program to indicate where the tester should start and stop. Also, the tester needs to be integrated with the bidirectional debugger that is in progress. The tester should be launchable from the debugger and may need to share information with it.

Parallelization of the tester could lead to tremendous speed improvements. Different branches of the depth–first search tree could be run in parallel since they do not communicate with each other. We have not investigated such parallelization.

Making the tester faster by optimizing its implementation will not yield as substantial performance improvements as modifying the algorithm, especially adding assumptions about which synchronized regions can be "trusted" (see Section 2.6).

Chapter 4

Results

This chapter presents the results of running the tester on a number of example client programs. Unfortunately Rivet does not yet support graphics. Since most multithreaded Java programs are applets or graphical applications, this severely limits the supply of real programs to test the tester on. Numerical analysis programs seem to be the most common non–graphical programs available, and they are not multithreaded. The programs presented here are toy programs, but they are scaled–down versions of what one would expect to encounter in real applications. They all have different types of concurrency errors that our tester correctly detects.

4.1 Performance

We will begin with a program that does not contain any errors but is constructed to measure how many paths (schedules) the tester must execute for a specified number of threads and locks per thread. The code for this Performance program is shown in Figure 4.1. The program takes two parameters: the first indicates how many threads to create, and the second how many locks to create. Each thread acquires each lock once and does nothing else. Thus there are no inter–thread data dependencies.

Results of running the program with various numbers of threads and locks per thread are listed in Table 4.1. Note that because there is an initial thread, having the program create two threads results in three running threads. For the executions taking under one minute, Rivet's nearly ten–second initialization time adds significant overhead that makes the paths per second number not very meaningful. The longer executions indicate that with

```java
/** Use this class to measure growth in number of paths */
public class Performance implements Runnable {
  static class Lock {}
  static Lock[] locks;
  static int numThreads;
  static int numLocks;

  public static void main(String[] args) {
    if (args.length != 2) {
      System.err.println("Arguments: <# threads> <# locks per thread>");
      return;
    }
    numThreads = Integer.valueOf(args[0]).intValue();
    numLocks = Integer.valueOf(args[1]).intValue();

    locks = new Lock[numLocks];
    for (int i=0; i<numLocks; i++)
      locks[i] = new Lock();

    for (int i=0; i<numThreads; i++)
      new Performance();
  }

  public Performance() {
    new Thread(this).start();
  }

  public void run() {
    for (int i=0; i<numLocks; i++)
      synchronized (locks[i]) {
      }
  }
}
```

Figure 4.1: The Performance program creates a specified number of threads and a specified number of locks per thread to measure how many paths the tester must search.

few locks per thread, the tester can execute up to about 80 paths per second. As more locks are added to each thread, this number decreases rapidly.

The large number of paths is due in part to the synchronized regions executed when the initial thread creates the other threads. By using the tester's `javasafe` option (see Section 3.3) we can eliminate most of this and focus only on synchronized regions involving the locks created by the Performance program itself. The right side of Table 4.1 gives numbers for testing runs that use the `javasafe` option, and Table 4.2 shows further results on numbers of threads difficult to test in a reasonable amount of time without the `javasafe` option. The option reduces the time dramatically for small numbers of locks; however, as the number of locks becomes large, the speedup disappears due the fact that the number of locks being ignored by `javasafe` becomes very small relative to the total number of locks acquired.

The number of paths executed for a test can differ from one platform to another. This is due to differences in the core JDK classes. For example, on JDK1.1.5, `java.lang.StringBuffer`'s `append` methods end up executing more synchronized regions than on JDK1.1.6. This causes the Performance 2 1 test to execute 16 paths on JDK1.1.5 versus the 10 paths reported in the table for JDK1.1.6. With the `javasafe` option the number of paths on the two platforms is identical, since the tester ignores the core JDK class' synchronized regions.

As the `javasafe` numbers indicate, in under an hour our tester can test a program containing eleven threads (in addition to the initial thread) if those threads only acquire one lock apiece. Real programs typically acquire more than one lock per thread, of course; the practical limit of our tester is five or six threads.

We now turn out attention to programs exhibiting errors. We will look at seven such programs. We summarize the performance results for these programs in Table 4.3. All numbers mentioned in the text of the rest of this chapter are for JDK1.1.6 on Windows NT on a Pentium 300MHz machine with 128MB RAM.

4.2 SplitSync

The SplitSync program was presented in Figure 1.1 on page 15; it is duplicated here as Figure 4.2, with an assertion inserted that will notify the tester when the invariant is false.

Program	normal			`javasafe`		
	Paths	Time	Paths/sec	Paths	Time	Paths/sec
Performance 2 1	10	00:00:12		4	00:00:12	
Performance 2 2	11	00:00:13		5	00:00:12	
Performance 2 3	12	00:00:13		6	00:00:12	
Performance 2 100	109	00:00:27		103	00:00:27	
Performance 2 250	259	00:02:05	2.1	253	00:02:12	1.9
Performance 2 500	509	00:09:44	0.9	503	00:10:09	0.8
Performance 2 1000	1009	00:46:01	0.4	1003	00:47:26	0.4
Performance 3 1	67	00:00:17		13	00:00:13	
Performance 3 2	85	00:00:17		21	00:00:12	
Performance 3 3	105	00:00:15		31	00:00:13	
Performance 3 50	3301	00:04:32	12.1	2757	00:03:38	12.6
Performance 3 100	11551	00:29:12	6.6	10507	00:27:53	6.3
Performance 4 1	546	00:00:19		40	00:00:13	
Performance 4 2	784	00:00:22		85	00:00:14	
Performance 4 3	1080	00:00:27		156	00:00:14	
Performance 4 10	5280	00:02:00	44.0	1885	00:00:51	
Performance 4 20	20800	00:12:09	28.5	11155	00:06:05	30.6
Performance 5 1	5447	00:01:16	71.7	121	00:00:14	
Performance 5 2	8632	00:01:59	72.5	341	00:00:16	
Performance 6 10	*778179	05:02:45	42.8	271453	01:45:06	43.0
Performance 7 5	*1025896	05:11:43	54.9	137257	00:37:04	61.7
Performance 8 3	*2356005	09:49:58	66.6	97656	00:21:18	76.4

Table 4.1: Number of paths (schedules) executed by the systematic tester on the Performance program of Figure 4.1 for various numbers of threads (first parameter) and locks per thread (second parameter). Each test was run both normally and with the `javasafe` tester parameter (see Section 3.3). A * means that the tester ran out of memory after the specified number of paths. The number of paths per second is only reported for tests lasting longer than one minute, since Rivet's 10 second initialization renders this statistic meaningless for shorter runs. These numbers are for JDK1.1.6 on Windows NT on a Pentium 300MHz machine with 128MB of RAM. The Rivet JIT was disabled.

Program	Tester Parameters	Paths	Time	Paths/second
Performance 5 3	javasafe	781	00:00:22	
Performance 5 4	javasafe	1555	00:00:33	
Performance 5 5	javasafe	2801	00:00:52	
Performance 5 10	javasafe	22621	00:07:44	48.8
Performance 6 1	javasafe	364	00:00:16	
Performance 6 2	javasafe	1365	00:00:27	
Performance 6 3	javasafe	3906	00:01:00	65.1
Performance 6 5	javasafe	19608	00:04:57	66.0
Performance 7 1	javasafe	1093	00:00:53	
Performance 7 2	javasafe	5461	00:01:18	70.0
Performance 7 3	javasafe	19531	00:04:20	75.1
Performance 8 1	javasafe	3280	00:00:48	
Performance 8 2	javasafe	21845	00:04:27	81.8
Performance 8 4	javasafe	335923	01:22:05	68.2
Performance 9 1	javasafe	9841	00:01:58	83.4
Performance 9 2	javasafe	87381	00:17:31	83.1
Performance 10 1	javasafe	29524	00:05:37	87.6
Performance 10 2	javasafe	349525	01:17:07	75.5
Performance 11 1	javasafe	88573	00:16:44	88.2
Performance 12 1	javasafe	265720	00:57:41	76.8

Table 4.2: Number of paths (schedules) executed by the systematic tester on the Performance program of Figure 4.1 for various numbers of threads (first parameter) and locks per thread (second parameter). The `javasafe` option is used to reduce the number of paths by causing the tester to ignore synchronized regions of the core JDK classes. The number of paths per second is only reported for tests lasting longer than one minute, since Rivet's 10 second initialization renders this statistic meaningless for shorter runs. These numbers are for JDK1.1.6 on Windows NT on a Pentium 300MHz machine with 128MB RAM. The Rivet JIT was disabled.

Program	Tester Parameters	Paths	Time	Paths/second
SplitSync		18	00:00:12	
NoEraser		16	00:00:12	
NoEraser	eraser	16	00:00:14	
Deadlock		10	00:00:13	
Deadlock	deadlock	841	00:00:22	
Deadlock3		79	00:00:13	
Deadlock3	deadlock	*207299	01:37:34	35.4
Deadlock3	deadlock, javasafe	2505	00:00:46	54.6
DeadlockWait		7	00:00:12	
DeadlockWait	deadlock	19	00:00:12	
BufferIf		22958	00:05:26	70.4
BufferIf	javasafe	169	00:00:15	
BufferIf	deadlock, javasafe	1781	00:00:37	
BufferNotify		29350	00:08:10	59.9
BufferNotify	javasafe	79	00:00:14	
BufferNotify	deadlock, javasafe	574	00:00:20	

Table 4.3: Number of paths (schedules) executed by the systematic tester on the seven example programs containing errors described in this chapter. A * means that the tester ran out of memory after the specified number of paths. The number of paths per second is only reported for tests lasting longer than one minute, since Rivet's 10 second initialization renders this statistic meaningless for shorter runs. These numbers are for JDK1.1.6 on Windows NT on a Pentium 300MHz machine with 128MB RAM. The Rivet JIT was disabled.

The assertion is in the form of a downcall and directly notifies the tester when it fails (downcalls are described in detail in Section A.4). This gives the tester the ability to notify the user of assertion failures and to stop testing when one occurs. Figure 4.3 shows a portion of the output from running the tester on SplitSync. The assertion violation is detected on the third path; a trace of that path is shown that lists the segments of each thread that were executed. When the tester is told not to stop at the first assertion failure, a total of 18 paths are explored in around 12 seconds, most of which is spent initializing Rivet; the same assertion failure occurs on six different paths.

4.3 NoEraser

The NoEraser program was presented in Figure 2.2; we repeat it here as Figure 4.4. For this program, Eraser by itself did not report any errors when running on our Java interpreter or on Rivet; running the tester and Eraser in parallel did catch the lock discipline violation, as shown in Figure 4.5. There are 16 paths in the program, and it takes about 14 seconds to test.

4.4 Deadlock

Figure 4.6 shows the code for a simple program that contains two threads with a potential lock–cycle deadlock. The tester correctly detects this deadlock, as Figure 4.7 shows. Without the `deadlock` parameter, the tester still finds the deadlock on the third of ten paths after about 11 seconds, most of which is spent initializing Rivet. When the `deadlock` parameter is used the deadlock is found 9 times in the 841 paths executed.

4.5 Deadlock3

The program in Figure 4.8 contains three threads that can enter a lock–cycle deadlock. The output is in Figure 4.9. Without the `deadlock` parameter, the tester finds the deadlock on path number 20 after 12 seconds, most of which is spent initializing Rivet. With the `deadlock` parameter, the deadlock is first found on path number 2790. Using both the `deadlock` and `javasafe` parameters, the deadlock is found on path number 122.

```java
public class SplitSync implements Runnable {
  static class Resource { public int x; }
  static Resource resource = new Resource();

  public static void main(String[] args) {
    new SplitSync();
    new SplitSync();
  }

  public SplitSync() {
    new Thread(this).start();
  }

  /** increments resource.x */
  public void run() {
    int y;
    synchronized (resource) { // A
      y = resource.x;
    }
    synchronized (resource) { // B
      // invariant: (resource.x == y)
      Downcall.downcall("Tester", "assert",
        new Object[]{new Boolean(x[0]==y),
                     "SplitSync.run: shared var was modified!"});
      resource.x = y + 1;
    }
  }
}
```

Figure 4.2: Sample Java program `SplitSync` illustrating a timing-dependent bug. By splitting the increment of `resource.x` into two `synchronized` statements, an error will occur if the two threads are interleaved between the `synchronized` statements. Both threads will read the original value of `resource.x`, and both will then set it to one plus its original value, resulting in the loss of one of the increments. This figure is identical to Figure 1.1 from Chapter 1, with the exception of the assertion inserted as a downcall.

```
!----------------------------------------------------!
Assertion violation: SplitSync.run: shared var was modified!
Call stack for thread Thread-0:
1: tools.tester.testsuite.SplitSync.run()V PC=70
0: <threadstart>.<ThreadStartMethod>()V PC=0
!----------------------------------------------------!

==> Aborted path # 3
-----------------------------------------
History of threads:
initial<start, class java.lang.Thread(1750)>
  Start= Call stack for thread initial:
0: <threadstart>.<StaticStartMethod>()V PC=0
  End  = Call stack for thread initial:
0: <threadstart>.<StaticStartMethod>()V PC=3

Thread-0<start, class tools.tester.testsuite.SplitSync$Resource(1726)>
  Start= Call stack for thread Thread-0:
0: <threadstart>.<ThreadStartMethod>()V PC=0
  End  = Call stack for thread Thread-0:
1: tools.tester.testsuite.SplitSync.run()V PC=15
0: <threadstart>.<ThreadStartMethod>()V PC=0

Thread-1<start, class java.lang.ThreadGroup(0)>
  Start= Call stack for thread Thread-1:
0: <threadstart>.<ThreadStartMethod>()V PC=0
  End  = Call stack for thread Thread-1:
0: <threadstart>.<ThreadStartMethod>()V PC=11

Thread-0<class tools.tester.testsuite.SplitSync$Resource(1726),
         class tools.tester.testsuite.SplitSync$Resource(1726)>
  Start= Call stack for thread Thread-0:
1: tools.tester.testsuite.SplitSync.run()V PC=15
0: <threadstart>.<ThreadStartMethod>()V PC=0
  End  = Call stack for thread Thread-0:
1: tools.tester.testsuite.SplitSync.run()V PC=73
0: <threadstart>.<ThreadStartMethod>()V PC=0
-----------------------------------------
```

Figure 4.3: A portion of the output from running the tester on the SplitSync program of Figure 1.1.

```
public class NoEraser implements Runnable {
  static class Resource { public int x; }
  static Resource resource = new Resource();

  // Two threads A and B
  // iff B writes resource.x before A -> Eraser data race
  public static void main(String[] args) {
    new NoEraser("A");
    new NoEraser("B");
  }

  private String name;

  public NoEraser(String name) {
    this.name = name;
    new Thread(this).start();
  }

  public void run() {
    int y;
    synchronized(resource) {
      y = resource.x;
    }
    if (y==0 && name.equals("B"))
      resource.x++;
    else synchronized(resource) {
      resource.x++;
    }
  }
}
```

Figure 4.4: A program that will pass Eraser's test nondeterministically. If the thread named `B` increments `resource.x` before `A`, `B` does so without obtaining the lock for `resource` first. This is a data race that Eraser will detect. However, if `A` increments `resource.x` before `B` does, Eraser will not report any errors. (This figure is a duplicate of Figure 2.2 on page 29.)

```
*** Data race detected on GETFIELD ***
        No lock consistently held while accessing:
(objID=1655, field=tools.tester.testsuite.NoEraser$Resource.x,
thread=("Thread-0" @ PC=77), READWRITE)
Call stack for thread Thread-1:
        1: tools.tester.testsuite.NoEraser.run()V PC=46
        0: <threadstart>.<ThreadStartMethod>()V PC=0

*** Data race detected on PUTFIELD ***
        No lock consistently held while accessing:
(objID=1655, field=tools.tester.testsuite.NoEraser$Resource.x,
thread=("Thread-0" @ PC=77), READWRITE)
Call stack for thread Thread-1:
        1: tools.tester.testsuite.NoEraser.run()V PC=51
        0: <threadstart>.<ThreadStartMethod>()V PC=0

*** Tester has completed (16 paths total). ***

*** Summary: Eraser found 12 data races. ***
```

Figure 4.5: A portion of the output from running the tester on the NoEraser program of Figure 2.2. It takes around 14 seconds to test the entire program.

4.6 DeadlockWait

The program in Figure 4.10 contains three threads that can enter a condition deadlock. The output is in Figure 4.11. The tester without the `deadlock` parameter finds the deadlock on the first path after about 11 seconds, most of which is spent initializing Rivet. With the `deadlock` parameter the number of paths increases from 7 to 19, and the deadlock is found quite a few times in those 19 paths.

4.7 BufferIf

The BufferIf program contains a bounded buffer that has an error in its dequeue method. Code for the buffer is in Figure 4.12. Any thread that attempts to enqueue when the buffer is full performs a `wait` on the buffer object; this puts the thread to sleep until another thread performs a `notify` or `notifyAll` on the same buffer object. Similarly, any thread dequeueing when the buffer is empty `waits` until the buffer has something in it. The error is that the buffer's dequeue method uses an `if` to test the condition that the buffer is full instead of a `while` loop. This means that if two threads are waiting for a full buffer to

```java
public class Deadlock implements Runnable {
  static class Lock {}
  static Lock a = new Lock();
  static Lock b = new Lock(); // the locks

  public static void main(String[] args) {
    new Deadlock("First", true);
    new Deadlock("Second", false);
  }

  boolean ab; // do we grab them in the order "ab"?

  public Deadlock(String name, boolean ab) {
    this.ab = ab;
    new Thread(this, name).start();
  }

  public void run() {
    if (ab) {
      synchronized (a) {
        synchronized (b) {
        }
      }
    } else {
      synchronized (b) {
        synchronized (a) {
        }
      }
    }
  }
}
```

Figure 4.6: A program that contains a potential deadlock. The first thread obtains the two locks in the order `a`,`b`, while the second thread obtains them in the order `b`,`a`. If the first thread obtains `a` and then the second thread obtains `b` we have reached a deadlock, since each thread holds the lock the other thread seeks.

```
!----------------------------------------------------------!
Deadlock detected -- a cycle between these threads:
Call stack for thread Thread-0:
        1: tools.tester.testsuite.Deadlock.run()V PC=21
        0: <threadstart>.<ThreadStartMethod>()V PC=0
(except move this thread back 1 opcode, to before most recent monitorexit)

Call stack for thread Thread-1:
        1: tools.tester.testsuite.Deadlock.run()V PC=60
        0: <threadstart>.<ThreadStartMethod>()V PC=0
!----------------------------------------------------------!

*** Tester has completed (10 paths total). ***
```

Figure 4.7: Portion of output from running the tester on the Deadlock program of Figure 4.6. It takes around 13 seconds to test the program.

become non–full, they could both wake up and both enqueue, overflowing the buffer, since they do not check upon waking up that the buffer is not full.

The program itself in Figure 4.13; it creates two producers and one consumer that share a single bounded buffer. One of the producers has a lower priority than the other producer and the consumer. A typical scheduler will thus never encounter the error since it will let the consumer run before letting the lower priority producer run. Our systematic tester finds the error, however — sample output from running the tester on the program is shown in Figure 4.14. The tester happens to find the error on the first path, which takes it about 11 or 12 seconds, most of which is spent initializing Rivet. The same condition deadlock is also detected on path numbers 3, 6, 11, 21, 31, and every 20 to 30 paths after that. Note that the error we find is a condition deadlock, but by inserting an assertion that the condition holds after the `if` check in the dequeue method we are guaranteed to find the problem (otherwise we would have to run with the `deadlock` option). Figure 4.14 shows that the assertion violation is detected at the same time as the deadlock.

4.8 BufferNotify

The BufferNotify program contains a bounded buffer very similar to that of the BufferIf program (see Section 4.7). The error in the BufferNotify program is that the buffer's enqueue and dequeue methods use `notify` instead of `notifyAll`. Code for the buffer is in Figure 4.15. The reason that `notifyAll` must be used is because there are actually two

```
public class Deadlock3 implements Runnable {
  static class Lock {}
  static Lock a = new Lock();
  static Lock b = new Lock();
  static Lock c = new Lock(); // the locks

  public static void main(String[] args) {
    new Deadlock3("First", 0);
    new Deadlock3("Second", 1);
    new Deadlock3("Third", 2);
  }

  int order; // code indicating lock grabbing order

  public Deadlock3(String name, int order) {
    this.order = order;
    new Thread(this, name).start();
  }

  public void run() {
    if (order == 0) {
      synchronized (a) {
        synchronized (b) {
        }
      }
    } else if (order == 1) {
      synchronized (b) {
        synchronized (c) {
        }
      }
    } else {
      synchronized (c) {
        synchronized (a) {
        }
      }
    }
  }
}
```

Figure 4.8: Another program containing a potential deadlock, this time between three threads. They each obtain two of the three locks in a different order. The deadlock occurs when the first thread grabs `a`, the second thread grabs `b`, and the third thread grabs `c`. Then none of the threads can proceed since a different thread holds the lock it seeks.

```
!---------------------------------------------------------!
Deadlock detected -- a cycle between these threads:
Call stack for thread Thread-0:
        1: tools.tester.testsuite.Deadlock3.run()V PC=21
        0: <threadstart>.<ThreadStartMethod>()V PC=0
(except move this thread back 1 opcode, to before most recent monitorexit)

Call stack for thread Thread-1:
        1: tools.tester.testsuite.Deadlock3.run()V PC=71
        0: <threadstart>.<ThreadStartMethod>()V PC=0
(except move this thread back 1 opcode, to before most recent monitorexit)

Call stack for thread Thread-2:
        1: tools.tester.testsuite.Deadlock3.run()V PC=110
        0: <threadstart>.<ThreadStartMethod>()V PC=0
!---------------------------------------------------------!

*** Tester has completed (79 paths total). ***
```

Figure 4.9: Portion of output from running the tester on the Deadlock3 program of Figure 4.8. It takes about 12 seconds to find this deadlock, 13 seconds to enumerate all paths of the program.

different conditions associated with the buffer object: that the buffer is not empty and that the buffer is not full. When a dequeue notifies, it means that the buffer is not full; when an enqueue notifies, it means that the buffer is not empty. However, it is possible that the thread that is woken up is waiting for the other condition. For example, suppose that one of the producers and the consumer of Figure 4.16 are waiting. Now the other producer performs an enqueue and the `notify` wakes up the first producer. This "consumes" the `notify`, preventing the consumer from being notified. Since both producers end up waiting, no one ever wakes up the consumer and we have a condition deadlock. The output from the tester illustrating the condition deadlock is shown in Figure 4.17. The tester finds the error on the first path after 11 seconds of execution, most of which is spent initializing Rivet. The same condition deadlock is detected on the third path, but then not again until path number 2346, followed by 2349, and then 3314 and 3317, with a similar pair every few thousand paths for the remaining paths.

Using `notifyAll` would solve the problem because then all threads would be woken up, and therefore the one that should be notified will actually get to run. `notifyAll` is needed only when a condition variable is associated with more than one condition.

```java
public class DeadlockWait implements Runnable {
  static class Lock {}
  static Lock a = new Lock();
  static Lock b = new Lock(); // the locks

  public static void main(String[] args) {
    new DeadlockWait("First", true);
    new DeadlockWait("Second", false);
  }

  boolean ab; // do we grab them in the order "ab"?

  public DeadlockWait(String name, boolean ab) {
    this.ab = ab;
    new Thread(this, name).start();
  }

  public void run() {
    if (ab) {
      synchronized (a) {
        synchronized (b) {
          try {
            b.wait();
          } catch (InterruptedException i) {
            System.out.println(name+" was interrupted!");
          }
        }
      }
    } else {
      synchronized (a) {
      }
      synchronized (b) {
        b.notify();
      }
    }
  }
}
```

Figure 4.10: A program containing a condition deadlock. The first thread obtains both locks `a` and `b` and then `waits` on `b`. The second thread then blocks trying to obtain lock `a`.

```
!---------------------------------------------------------!
warning, not all threads died -- 1 thread is waiting,
 and 1 thread is blocked on a lock!
!---------------------------------------------------------!
==> Finished path # 0
------------------------------------------
History of threads:
initial<start, class java.lang.Thread(1677)>
  Start=        Call stack for thread initial:
                0: <threadstart>.<StaticStartMethod>()V PC=0
  End  =        Call stack for thread initial:
                0: <threadstart>.<StaticStartMethod>()V PC=3

Thread-1<start, class java.lang.ThreadGroup(0)>
  Start=        Call stack for thread Thread-1:
                0: <threadstart>.<ThreadStartMethod>()V PC=0
  End  =        Call stack for thread Thread-1:
                0: <threadstart>.<ThreadStartMethod>()V PC=11

------------------------------------------
*** Tester has completed (7 paths total). ***
```

Figure 4.11: Portion of output from running the tester on the DeadlockWait program of Figure 4.10. It takes about 12 seconds to test the program.

`notifyAll` is inefficient, since probably only one of the woken threads will be able to do anything; the others will just wait again. Of course the inefficiency only happens when many threads are waiting.

4.9 Summary

This chapter has shown that our tester is capable of finding many types of common errors in multithreaded programs that are often difficult to test for. These include lock–cycle deadlocks, condition deadlocks, using an `if` instead of a `while` to test a condition variable, using `notify` instead of `notifyAll` to wake up threads waiting on a condition variable that is used for more than one type of thread, mutual–exclusion locking discipline violations that do not occur in all dynamic schedules, and errors that occur only in certain orderings of modules that are individually correctly synchronized.

```java
/** Shared, bounded buffer */
public class Buffer {
  static final int BUFSIZE = 2; // so capacity = 1
  private int first, last;
  private Object[] els;

  public Buffer() { first = 0; last = 0; els = new Object[BUFSIZE]; }

  public synchronized void enq(Object x) throws InterruptedException {
    if ((last+1) % BUFSIZE == first) // error -- should be while, not if
      this.wait();
    // invariant: (last+1) % BUFSIZE != first
    els[last] = x;
    last = (last+1) % BUFSIZE;
    this.notifyAll();
  }

  public synchronized Object deq() throws InterruptedException {
    while (first == last)
      this.wait();
    Object val = els[first];
    first = (first+1) % BUFSIZE;
    this.notifyAll();
    return val;
  }
}
```

Figure 4.12: Sample bounded buffer class containing a timing–dependent error. The `enq` function should use a `while` instead of an `if`. With an `if`, if the enqueueing thread is woken up but some other thread executes before it takes control and changes the condition, the enqueueing thread will go ahead and execute even though the condition is false. Consider the program in Figure 4.13 that uses this buffer. Assume both producers are waiting on a full buffer and the consumer dequeues one item. This notifies both producers. If one runs and enqueues an item, filling up the buffer, and then the other runs it will enqueue an additional item, overflowing the buffer.

```
/** Producer-consumer sharing a bounded buffer */
public class BufferIf {
  static final int ITEMS_PRODUCED = 2;

  public static void main(String[] args) {
    Buffer b = new Buffer();
    new Thread(new Producer(b, "P1")).start();
    new Thread(new Consumer(b)).start();
    Thread p2 = new Thread(new Producer(b, "P2"));
    p2.setPriority(Thread.MIN_PRIORITY);
    p2.start();
  }

  static class Producer implements Runnable {
    private Buffer buffer;
    private String name;
    public Producer(Buffer b, String n) { buffer = b; name = n; }
    public void run() {
      try {
        for (int i=0; i<ITEMS_PRODUCED; i++)
          buffer.enq(name);
      } catch (InterruptedException i) { System.err.println(i); }
    }
  }

  static class Consumer implements Runnable {
    private Buffer buffer;
    public Consumer(Buffer b) { buffer = b; }
    public void run() {
      try {
        for (int i=0; i<ITEMS_PRODUCED*2; i++)
          buffer.deq();
      } catch (InterruptedException i) { System.err.println(i); }
    }
  }
```

Figure 4.13: Program that creates two producers and one consumer that share a bounded buffer. One producer has lower priority than the other threads. This means that a typical scheduler will never encounter the `if` versus `while` bug in Figure 4.12, since that bug depends on both producers executing before the consumer.

```
!-----------------------------------------------------!
Assertion violation: Buffer.enq: buffer overflow!
Call stack for thread Thread-0:
        2: tools.tester.testsuite.BufferIf$Buffer.enq(Ljava/lang/Object;)V PC=62
        1: tools.tester.testsuite.BufferIf$Producer.run()V PC=13
        0: <threadstart>.<ThreadStartMethod>()V PC=0
!-----------------------------------------------------!

!------------------------------------------------------!
warning, not all threads died -- 1 thread is waiting,
 and 0 threads are blocked on locks!
!------------------------------------------------------!

==> Finished path # 0
-----------------------------------------
History of threads:
initial<start, class java.lang.Thread(1683)>
Thread-0<start, class tools.tester.testsuite.BufferIf$Buffer(1655)>
Thread-1<start, class tools.tester.testsuite.BufferIf$Buffer(1655)>
Thread-2<start, class tools.tester.testsuite.BufferIf$Buffer(1655)>
Thread-0<class tools.tester.testsuite.BufferIf$Buffer(1655),
         class java.lang.ThreadGroup(0)>
Thread-1<class tools.tester.testsuite.BufferIf$Buffer(1655),
         class tools.tester.testsuite.BufferIf$Buffer(1655)>
Thread-2<class tools.tester.testsuite.BufferIf$Buffer(1655),
         class java.lang.ThreadGroup(0)>
Thread-1<class tools.tester.testsuite.BufferIf$Buffer(1655),
         class tools.tester.testsuite.BufferIf$Buffer(1655)>
-----------------------------------------
```

Figure 4.14: Portion of output from running the tester on the BufferIf program of Figures 4.12 and 4.13. It takes about 12 seconds for the tester to find this error. Since a full trace of this path showing the method stack for the start and end point of each thread segment would take far too much room, a tracing option was used that summarizes thread segments by the classes of the locks whose synchronized regions are boundaries of the segments. This is an ambiguous representation since the `Buffer` object is used as a lock by both producers and the consumer, but this trace still gives a good idea of what happened. The threads are numbered in the order they are created. Thus the trace shows that the order of execution was Producer1, Consumer, Producer2, Producer1 (dies), Consumer, Producer2 (dies), Consumer. The fact that two producers executed after one another is an indication that the buffer may have overflowed; this resulted in the consumer missing an item, which is why it ends up waiting at the end.

```
/** Shared bounded buffer */
public class Buffer {
  static final int CAPACITY = 1;
  // Need extra slot to tell full from empty
  static final int BUFSIZE = CAPACITY+1;
  private int first, last;
  private Object[] els;

  public Buffer() { first = 0; last = 0; els = new Object[BUFSIZE]; }

  public synchronized void enq(Object x) throws InterruptedException {
    while ((last+1) % BUFSIZE == first)
      this.wait();
    els[last] = x;
    last = (last+1) % BUFSIZE;
    this.notify(); // error -- should be notifyAll()
  }

  public synchronized Object deq() throws InterruptedException {
    while (first == last)
      this.wait();
    Object val = els[first];
    first = (first+1) % BUFSIZE;
    this.notify(); // error -- should be notifyAll()
    return val;
  }
}
```

Figure 4.15: Sample bounded buffer class containing a timing–dependent error. The `enq` and `deq` functions should use `notifyAll` instead of `notify`.

```java
/** Producer-consumer sharing a bounded buffer */
public class BufferNotify {
  static final boolean OUTPUT = false;
  static final int ITEMS_PRODUCED = 2;

  public static void main(String[] args) {
    Buffer b = new Buffer();
    new Thread(new Producer(b, "P1")).start();
    Thread p2 = new Thread(new Producer(b, "P2"));
    p2.setPriority(Thread.MIN_PRIORITY);
    p2.start();
    new Thread(new Consumer(b)).start();
  }

  static class Producer implements Runnable {
    private Buffer buffer;
    private String name;
    public Producer(Buffer b, String n) { buffer = b; name = n; }
    public void run() {
      try {
        for (int i=0; i<ITEMS_PRODUCED; i++) {
          buffer.enq(name);
        }
      } catch (InterruptedException i) { System.err.println(i); }
    }
  }

  static class Consumer implements Runnable {
    private Buffer buffer;
    public Consumer(Buffer b) { buffer = b; }
    public void run() {
      try {
        for (int i=0; i<ITEMS_PRODUCED*2; i++) { // while (true)
          buffer.deq();
        }
      } catch (InterruptedException i) { System.err.println(i); }
    }
  }
```

Figure 4.16: Program that creates two producers and one consumer that share a bounded buffer. One of the producers has lower priority than the other threads. This means that a typical scheduler will never encounter the `notify` versus `notifyAll` bug in the buffer of Figure 4.15, since that bug depends on a producer being notified in preference to a consumer.

```
!--------------------------------------------------------!
warning, not all threads died -- 2 threads are waiting,
 and 0 threads are blocked on locks!
!--------------------------------------------------------!

==> Finished path # 0
-----------------------------------------
History of threads:
initial<start, class java.lang.Thread(1685)>
Thread-0<start, class tools.tester.testsuite.BufferNotify$Buffer(1655)>
Thread-1<start, class tools.tester.testsuite.BufferNotify$Buffer(1655)>
Thread-2<start, class tools.tester.testsuite.BufferNotify$Buffer(1655)>
Thread-0<class tools.tester.testsuite.BufferNotify$Buffer(1655),
         class java.lang.ThreadGroup(0)>
Thread-1<class tools.tester.testsuite.BufferNotify$Buffer(1655),
         class tools.tester.testsuite.BufferNotify$Buffer(1655)>
-----------------------------------------
```

Figure 4.17: Portion of output from running the tester on the BufferNotify program of Figures 4.15 and 4.16. The tester detects the deadlock after 11 seconds of execution. Since a full trace of this path showing the method stack for the start and end point of each thread segment would take far too much room, a tracing option was used that summarizes thread segments by the classes of the locks whose synchronized regions are boundaries of the segments. This is an ambiguous representation since the `Buffer` object is used as a lock by both producers and the consumer, but this trace still gives a good idea of what happened. The threads are numbered in the order they are created. Thus the trace shows that the order of execution was Producer1, Producer2, Consumer, Producer1 (dies), Producer2. Producer1's last execution's `notify` woke up Producer2 instead of the consumer, and we end up with both Producer2 and the Consumer waiting.

Chapter 5

Conclusions

Conventional testing methods are inadequate for multithreaded Java programs. Due to the inherent nondeterminism of multithreaded programs, testing a program by merely running it on different inputs cannot guarantee to cover all program behaviors. For this a behavior–complete tester is needed. Such testers exist for multi–process programs, but their methods do not apply well to the shared address space of multithreaded Java programs.

We have shown that it is possible to build a systematic, behavior–complete tester for multithreaded Java programs by making assumptions about the nature of the programs. The fundamental assumption is that the program to be tested follows a mutual–exclusion locking discipline. Given this assumption we need only enumerate all schedules of synchronized regions instead of all schedules of instructions to cover all behaviors of the program. This assumption is not overly restrictive: as Savage et al. [S+97] argue, even experienced programmers tend to follow such a discipline. They do this even when more advanced sychronization techniques are readily available.

We presented the ExitBlock algorithm that guarantees to test all program behaviors while only considering the possible schedules of synchronized regions. Our ideas are applicable to other multithreaded languages. The only Java–specific properties used by the ExitBlock algorithm are that threads own locks and that locks are properly nested. We proved the algorithm correct and we have shown that our implementation correctly finds the various errors in seven example programs in a short amount of time. These errors include common multithreading errors that are often difficult to test for, such as errors that occur only in certain orderings of modules that are individually correctly synchronized, or using an `if` instead of a `while` to test a condition variable.

While our tester cannot currently test larger programs due to its slow speed, it is built on a prototype virtual machine and is intended as a proof of concept. Implementing a similar tester in a high–performance virtual machine will enable testing a much larger, more practical set of target programs.

The hardest challenges in implementing the ExitBlock algorithm are checkpointing and deterministic replay. We benefited from placing these inside of the virtual machine. Because they must intercept low–level operations (every field access and input or output action) they need to be in direct contact with those operations for efficiency.

We are satisfied with the power that Rivet's Tool Interface gives tool writers. It enabled rapid development of Eraser and the tester; implementing the tester independently would have taken significantly longer.

Appendix A

Rivet Implementation

The Rivet Virtual Machine is a Java virtual machine designed as a platform for tools. Rivet supports tools by exporting client program data and execution events, along with an extra field mechanism, deterministic replay, and checkpointing facilities that allow a tool to undo client program operations.

Our systematic tester makes heavy use of Rivet's checkpointing, extra fields, and event system. The tester's deterministic replay is not yet implemented but it will build on Rivet's. This appendix describes the implementation of Rivet, focusing on those features used by the tester.

A.1 Architecture Overview

Rivet is a Java virtual machine written in Java. It is implemented in Java for modularity and extensibility, at some expense to performance (performance numbers for Rivet can be found in Section 3.1.1). Since it runs on top of another Java virtual machine, Rivet can avoid platform–specific code by utilizing features of the "lower" virtual machine.

Writing a Java virtual machine in Java has some inherent difficulties. The implementors of the *JavaInJava* virtual machine [Tai98], also written in Java, discovered a number of such challenges. We experienced some similar problems to theirs, but took different approaches on many issues.

A Java virtual machine is typically composed of several subsystems, including a class loader, bytecode interpreter, memory manager, and thread scheduler. A detailed description of what a Java virtual machine is can be found in [MD97] or in the official Java virtual

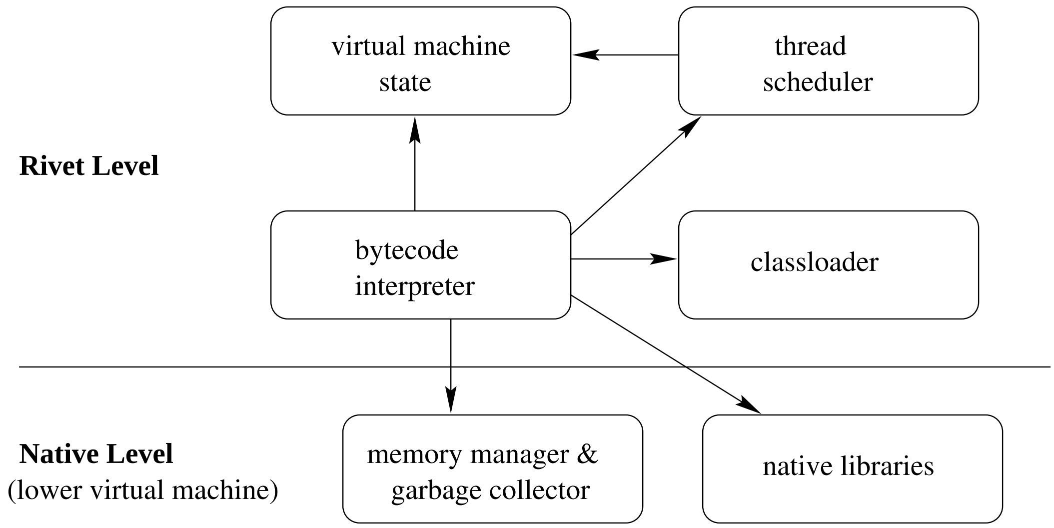

Figure A.1: The major subsystems of Rivet. Rivet makes use of the lower virtual machine's memory manager, garbage collector, and native libraries.

machine specification [LY99]. Figure A.1 illustrates the major subsystems of Rivet. The design of Rivet cleanly separates these subsystems and uses the class and protection mechanisms of Java to enforce modularity and communication through established interfaces. This is in contrast to most virtual machines whose main priority is performance and which tend to sacrifice a clean, modular design in the name of efficiency. JavaInJava joins Rivet in the category of virtual machines that consider clean design to be as important as performance. Rivet does relax its interfaces in the performance–critical bytecode interpretation loop, allowing direct access to member variables between critical classes.

Rivet and JavaInJava also share a common approach to memory management. Because pointers are not available in Java, virtual machine memory management would be very different if written in Java than if written in C or C++. Rivet leaves all memory management issues to the underlying virtual machine, relying on its garbage collector and memory allocation schemes. Since Rivet has no direct control over the garbage collector it must be careful not to generate too many internal structures that can reference client program objects that should be garbage collected; doing so would prevent garbage collection. Weak references would be a big help here. At the moment Rivet simply avoids such structures.

A.2 Class and Object Representation

Java classes are loaded dynamically as needed by a Java virtual machine. The machine needs some internal representation of a class. Rivet uses the `DE.fub.inf.JVM.JavaClass` package [Dah99] to parse and manipulate class files. Rivet uses the JavaClass representation of the constant pool as is; however, Rivet has its own structures that represent classes, methods, fields, and exception tables. The design of Rivet calls for each JavaClass class to be used only by the corresponding Rivet class, keeping dependence on the JavaClass package to a minimum so that it can be replaced in the future if need be.

Figure A.2 illustrates class representation in Rivet. A module called RClassLoader creates an RClass instance for each class. RClass contains method and field information. We will not discuss how methods are stored here as it is not relevant to Rivet's relationship with the tester. Fields are stored in different ways depending on what client object representation is being used. Rivet currently does not do any verification of classes that it loads.

Client objects are instances of client classes. Rivet supports multiple representations of client objects. It hides object representation specifics behind a series of "Rep" interfaces. General operations on objects, such as obtaining the RClass of an object or cloning an object, are provided by the RepInterface interface. The RepArrayAccess interface allows for array manipulation, much like the `java.lang.reflect.Array` class. The RepClass interface knows how to create a new instance of its class and holds the field accessors for its class in the form of RepFields. These RepFields are what the rest of the virtual machine uses to read and write fields of client objects. As we shall see, this abstraction is of great value in implementing checkpointing and extra fields.

After the RClassLoader creates an RClass, the RClass requests that the RepClassLoader create a corresponding RepClass. RepClassLoader is needed because some object representations perform extra operations during class loading.

Two object representations will be discussed here, the Generic representation and the Native representation. Only the Native one is used substantially because the other representations do not work with native methods.

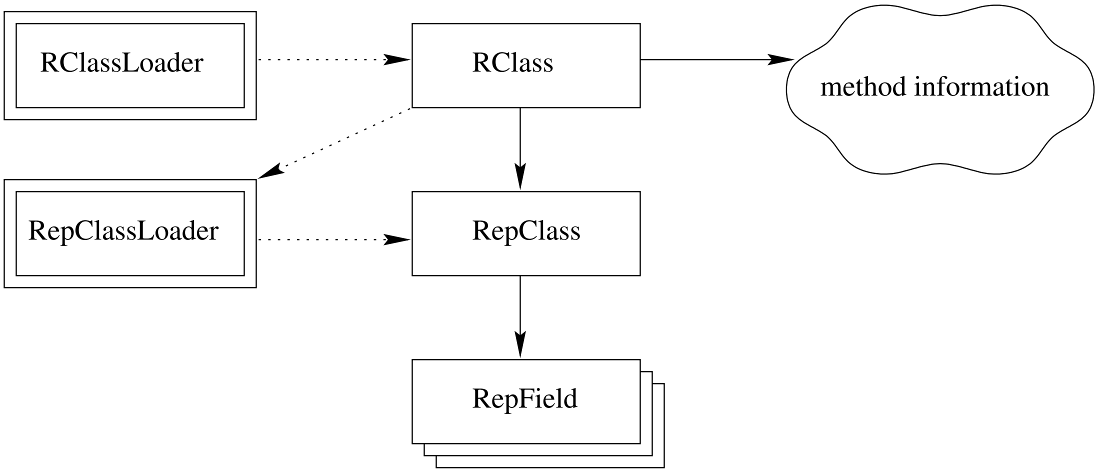

Figure A.2: Client class representation in Rivet. Dotted lines indicate creation: when a class is loaded, the RClassLoader creates an RClass. The RClass then tells the RepClassLoader to create a RepClass; this RepClass is stored with the RClass, as indicated by the solid arrow. Solid arrows also show that information about the class' methods (whose exact representation will not be discussed here) are stored in the RClass, and that the RepClass contains information specific to the object representation being used, such as field accessors (RepFields).

A.2.1 Generic Representation

The Generic representation is a simple, straightforward representation: it represents a client object as an instance of Rivet's GObject class. Each GObject contains the fields for its corresponding client object and a reference to the RClass for its client object's class. Because Java has no polymorphism, the fields of the client object are stored in three separate arrays. Primitives of integral types (`byte`, `char`, `short`, `int`, and `long`) are stored in an array of `int`s. `Long`s are split into their high and low words, taking up two slots. Primitives of type `float` are widened to `double`s and stored along with `double`s in an array of `double`s. Fields of reference type (arrays and objects) are stored in an array of GObject. Figure A.3 summarizes the fields of GObject. This differs from JavaInJava's solution, which wraps primitives in their corresponding object wrappers (`int` to Integer, etc.) and then deals with them all as Objects. Rivet took its approach for efficiency.

The Generic representation works fine except for dealing with native methods. We would like to avoid duplicating the substantial native libraries (for example, the core of `java.io` and `java.awt` are implemented natively) of the underlying virtual machine for Rivet. If Rivet can reuse the underlying virtual machine's native methods, then Rivet

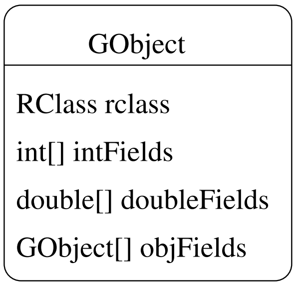

Figure A.3: Fields of GObject, the class that represents a client object in the Generic representation. Three arrays must be used to hold the fields of the client object to get around Java's lack of efficient polymorphism.

will be much more platform–independent. However, in the Generic representation, a client object at runtime is represented by an instance of class GObject, no matter what client class the object is an instance of, which prevents using the existing native methods.

Consider class `java.io.FileDescriptor`. The native code that manipulates FileDescriptors expects to be handed an instance of FileDescriptor that is represented in the way the lower virtual machine represents objects. If Rivet tries to invoke some native method on a GObject, the method will fail since it will not find the fields of FileDescriptor in the proper place. This is illustrated in Figure A.4. In order to reuse lower native methods Rivet must use the same object representation as the lower virtual machine. This realization led to the creation of the Native object representation.

A.2.2 Native Representation

As described in the previous section, the native libraries of the lower virtual machine expect client objects to be in the form that the lower virtual machine represents them in. The namespace of Java classes is delineated not just by class name but by class name and class loader. Due to a loophole in the current Java specifications, it turns out that it is possible to load a class B with the same name as class A through a different classloader than class A and have B share A's native methods. This is because native method linking does not involve looking at a class' classloader, although the rest of the system differentiates classes with the same name coming from different classloaders. Rivet exploits the hole in its Native representation.

The Native representation's implementation of RepClassLoader is called NClassLoader. For every client class C loaded by Rivet, the NClassLoader loads a shadow class

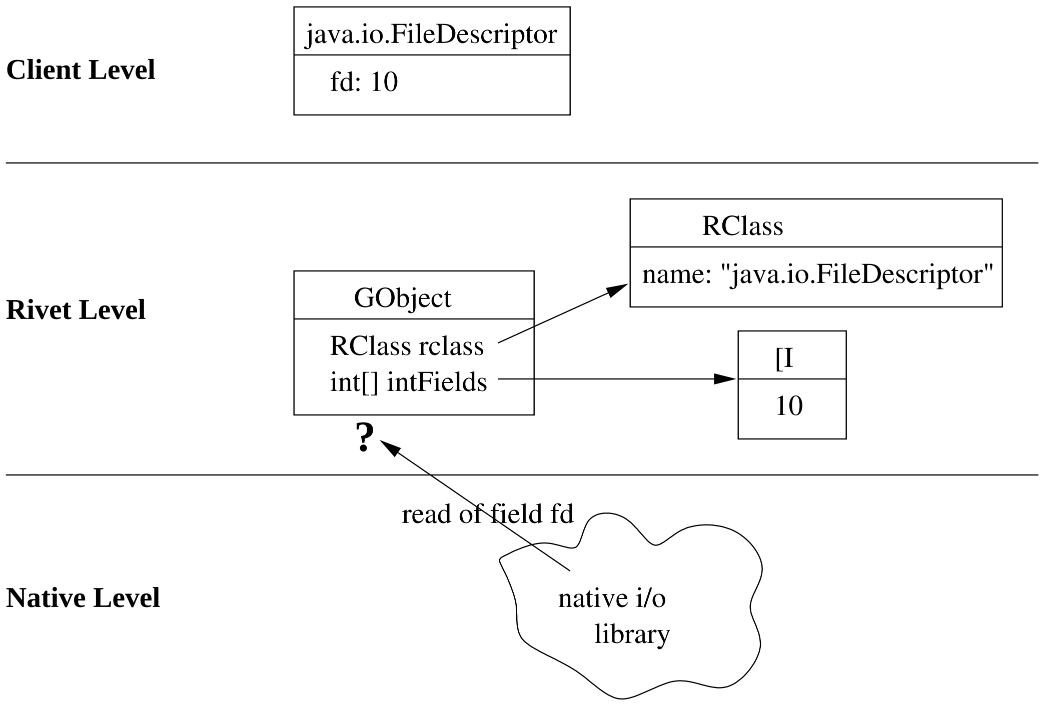

Figure A.4: Problem with invoking lower native methods using the Generic object representation. The Client Level is the level of the client program, the Rivet Level is that of the representations used by Rivet, and the Native Level is the level of the lower virtual machine. The problem is that native code expects client objects to be in the same format as that used by the lower virtual machine.

C' that shares C's native methods. An instance of this shadow class is created as the representation of a client instance of class C. This shadow class instance has fields in the proper places for the native methods of the client class. Figure A.5 illustrates the new situation. Lower virtual machine native methods may now be directly invoked on client objects, which are instances of shadow classes. Note that the shadow class can have more fields and methods than the client class, and that its fields and methods can be slightly different, so long as native code will not notice. For example, its fields can have different access modifiers since native code does not check those.

Implementation of Native Representation

There are several challenges in implementing this scheme, including how to give Rivet access to client object fields. The Java Reflection API must be used, but it does not allow access to non–public fields. Also, Rivet needs a way to obtain the RClass for a client object. Another issue is how to handle upcalls (invocations of Java methods by native methods) in

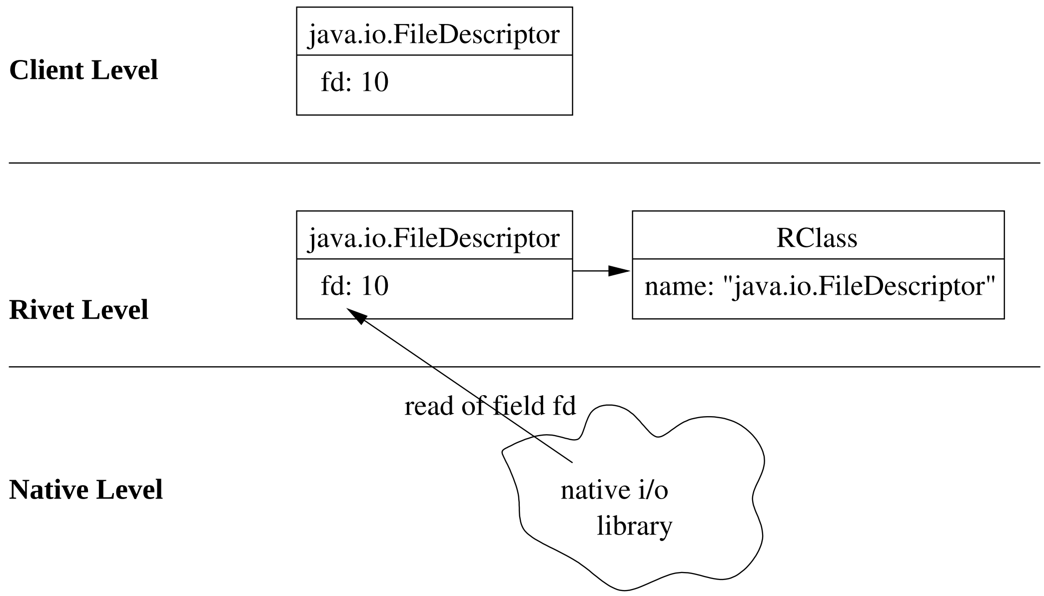

Figure A.5: The Native representation makes it possible to directly invoke lower native methods on Rivet's representation of client objects. It uses shadow classes that have the same name but were loaded by a different class loader than the corresponding client class. In the figure, the Rivet–level FileDescriptor is an instance of the shadow class for `java.io.FileDescriptor`. This means that native code will access the correct `fd` field when presented with a Rivet–level instance.

this scheme. All of these problems can be solved by dynamically transforming the shadow class.

The NClassLoader can modify the shadow class while keeping the properties of it that native method invocation needs, namely, that it has the same name as the client class and that the representation of instances of it in the lower virtual machine look sufficiently like instances of the client class to lower native methods. For native methods that use the Java Native Interface (JNI), this means that the shadow class simply has to duplicate the fields and methods of the client class in name and type. Some core JDK native methods use direct field offsets instead of the JNI; for these we need shadow classes to have the same memory layout as client classes. This is discussed further in Section A.5.

To solve the problem of the Reflection API not allowing access to non–public fields, we make every field of the shadow class public. As for obtaining the RClass given a client object, we initially added a static field to every class. However, given a client object the new field must be looked up every time with a call to `java.lang.Class.getField`, which is inefficient. Instead the Native representation keeps a hashtable mapping client classes to

RepClasses, and from the RepClass the RClass can be obtained.

Upcalls are invocations of Java methods by native methods. An example is the creation of a Java object by native code that leads to execution of the object's constructor. Since Rivet should interpret all Java methods, we need to make sure that if a native method makes an upcall Rivet knows about it. We can do this by replacing each of a shadow class' non–native methods with a *trampoline*. This trampoline calls a special Rivet static method (the upcall handler) and passes it the name of the Java method that the native method is trying to invoke. The upcall handler then invokes the Rivet interpreter on that Java method. Note that this trampoline transformation means that the parent of a shadow class needs to be the shadow class corresponding to its client class' parent, not its client class' parent itself. This is because inherited methods need to be trampolines as well. Thus we must have a complete shadow hierarchy.

There is another transformation that must be made. In Java there is a separation between memory allocation for a new object and initialization of that memory. The `new` opcode allocates memory, while constructors initialize it. With the Native representation, in order to allocate memory we must call a constructor of the shadow class in question. We do not want this constructor to do any initialization in order to keep allocation separate. Thus we must add a dummy constructor to every shadow class that does no initialization. This also constrains us to have shadow classes inherit from shadow classes, since these dummy constructors must call corresponding dummy constructors in their superclasses.

All of these transformations are performed in Rivet using the ClassGen package of JavaClass, which allows dynamic manipulation of Java class files. The NClassLoader is a custom classloader that transforms each client class into a shadow class that is then registered with the lower virtual machine.

There is one more problem: the verifier of the lower virtual machine will not allow a shadow class of `java.lang.Object` or of `java.lang.Throwable`. Thus we must share these classes with the lower virtual machine. They become special cases. Every chain of dummy constructor calls always ends with a call to Object's constructor, which does no initialization anyway and so functions as a dummy constructor, or Throwable's no–argument constructor, which does a little initialization. Throwable's minor initialization is something that we can live with. However, Throwable contains some private fields. Thus Rivet requires that a version of Throwable with only public fields be placed before `classes.zip` on the classpath.

Array classes must also be shared with the lower virtual machine. Since array classes are a special case in Java anyway (they are only created by the virtual machine, and do not go through class loaders), and since they do not have private fields or native methods, they are not much of a problem here. However, they will be more of a problem when we consider extra fields and checkpointing later on.

As for interfacing to the rest of Rivet, the RepFields hide the Reflection API calls, and client objects can still be thought of as instances of Object — whether they are in fact instances of shadow classes or instances of GObject.

Consequences of Native Representation

The Native representation gives Rivet the ability to reuse the lower virtual machine's native methods. However, it has a number of limitations. Performance is one: reflection is many times slower than the array access needed for the Generic representation (Rivet's JIT solves this problem, though, by replacing this reflection with direct field access). Another is that some native methods construct client objects without calling their Java–code constructors. Thus Rivet is not notified that these objects have been created. Typically they are the return value of the native method, and typically it is a String that is created, so Rivet has a special–case check to handle this properly. Additionally, in order to be able to convert from a lower virtual machine object to a Rivet client object, the class of the lower object must have all public fields. This can be accomplished by using a `classes.zip` in which all fields are public.

Another limitation is the fragility of Rivet during a native upcall. Thread switches during an upcall are disastrous (see Section A.3), and checkpoints cannot be taken (see Section A.6.4). Upcalls do tend to be rare, fortunately.

Even more limitations show up when we add extra fields and checkpointing; they are described in those features' respective sections (A.5 and A.6.4). Fortunately, client application native methods are typically well–behaved. It is usually only the core JDK native methods that behave in ways that Rivet cannot handle well, and these can be dealt with on a case–by–case basis.

The JavaInJava virtual machine uses a client object representation similar to Rivet's Generic representation. Like Generic, JavaInJava cannot reuse native methods. JavaInJava found no acceptable solution to the problem of native methods in a virtual machine written entirely in Java.

A.3 Interpretation

The heart of Rivet is the bytecode interpreter. This consists of a series of loops called `run`, `runThreadForAWhile`, and `runMethodForAWhile`. The `run` loop calls `runThreadForAWhile`, which returns when a thread switch is desired, at which point `run` performs the thread switch and then calls `runThreadForAWhile` again. The `runThreadForAWhile` loop calls `runMethodForAWhile` on the current method of the current thread, which returns when that method completes or when a new method invocation is being made. `runThreadForAWhile` then calls `runMethodForAWhile` on the new current method. `runMethodForAWhile` is a loop containing a large switch statement that interprets the bytecodes of the current method.

The representation of threads is influenced by both the extra fields feature of Rivet and the checkpointing requirements. Each client thread has a corresponding internal class that stores the thread's activation stack, its PC (program counter), and its scheduling information. An extra field is added to the client–level `java.lang.Thread` (see Section A.5 for information on how this is done) to refer to the internal class for that thread. Instead of using a direct reference, an index into the scheduler's array of all threads is stored in the extra field. This level of indirection eases checkpointing of the thread scheduler. This will be discussed further in Section A.6.2.

Each thread's activation stack is a stack of frames, one frame per method invocation. The thread's PC indicates which bytecode is the next to be executed in the method invocation on top of the stack. Each frame contains a PC to be returned to when the method it invoked returns, an operand stack, and a list of local variables. The same method as in the Generic representation is used here to get around the lack of polymorphism in Java: both the operand stack and the local variables are kept in three separate arrays.

Rivet consists of a single thread. Having only one thread running on the lower virtual machine makes Rivet's task of deterministic replay much simpler, and makes the implementation of the thread scheduler much simpler. All client threads are multiplexed on Rivet's single thread. A global counter of bytecodes executed is used to decide when to preempt a thread. Note that Rivet's implementation of multithreading is inherently platform–independent.

The problem with having a single Rivet thread is that Rivet cannot handle thread switches in native upcalls. An upcall involves stacking interpreters on top of each other on the lower virtual machine's method stack. Imagine a thread switch during an upcall in which the thread switched to itself invokes a native method that makes an upcall. If Rivet switches back to the first thread, it must destroy the second thread's native method frame on the lower virtual machine's stack in order to return to the first thread's native method. This problem could be solved by having a separate lower virtual machine thread execute each native call (which would be inefficient), or having a separate lower virtual machine thread for each client thread. This has not been addressed yet mainly because native upcalls are rare and because Rivet cannot handle checkpoints during them anyway (see Section A.6.4). Currently, Rivet makes no promises about working properly with native upcalls that perform complex operations.

For input and output, a special thread is launched for every potentially blocking operation. If this thread blocks, the scheduler blocks the current client thread and switches to a new thread. Every so often the scheduler walks through all threads that are blocked on input or output to check if they are now ready to run. Note that since we intercept only those native methods in the core JDK that can block, client native methods must be non–blocking in order for the scheduler to operate properly. This requirement could be removed in the same manner proposed earlier to eliminate problems with thread switches in upcalls, by having a separate thread execute each native method call. Again, this would be inefficient.

Synchronization is managed by adding an extra field to the client–level `java.lang.Object` that holds the index into a master list of lock data structures. Each lock holds a reference to the thread that owns it, a count of how many times it has been acquired by that thread, and two queues: one of threads blocked waiting for that lock and one of threads waiting to be notified on that lock. As with threads, the index indirection helps with checkpointing of the lock structures.

A Just In Time dynamic compiler (JIT) for Rivet is under development. It translates client program bytecodes to bytecodes that execute directly on the lower virtual machine. Since the lower machine's own JIT already provides extensive optimizations, Rivet's JIT can avoid performing expensive analysis and immediately pass the bytecodes to the lower machine.

A.4 Tool Interface

The purpose of Rivet is to implement common features that Java programming tools require and make them available for rapid tool development. The Rivet Tool Interface, or RTI, is a set of interfaces used by tools to access Rivet's powerful features.

One group of RTI interfaces mirrors the Java Reflection API, providing client object reflection capabilities to tools. The RTIClass, RTIMethod, RTIConstructor, RTIField, and RTIArray interfaces resemble their Java Reflection counterparts. The RTIObject interface contains one method allowing a tool to obtain the RTIClass of a client object. Once the tool has this it can use the other reflection interfaces to get and set fields and invoke methods.

The RTIClassLoader interface contains methods for looking up classes by name and loading new classes. In addition, it allows a tool to add extra fields to a class (this must be done before any classes are loaded). This powerful feature enables tools to directly store data in client objects. Section A.5 describes the extra field mechanism in detail.

The RTIFrame and RTIThread interfaces allow a tool to view the current threads and their activation stacks, and the operand stacks and local variables of each activation frame. Threads may be disabled using RTIThread; a thread so disabled will not run or be notified (unless there are no alternative threads to notify) until a tool re–enables it.

The namesake interface, RTI, is the main interface. It contains access methods for the RTIClassLoader corresponding to the system loader and the singleton RTIArray and RTIObject classes (since they are interfaces they cannot have static methods, so there must be one instance of each of them). It has methods to get the live threads, get the threads blocked on a lock, and get the owner of a lock. In addition it has methods to set and delete breakpoints and watchpoints, methods for taking and returning to checkpoints, and methods for registering for events. Checkpoints and events are the two most important features of the RTI. We will discuss events now and checkpoints in Section A.6.

Events allow a tool to be notified when certain operations occur. A tool registers an event handler for each event it wishes to receive. There are separate events for every opcode, which are generated whenever a bytecode of that type is executed. There are events generated whenever a class is loaded, whenever an object is created, an whenever an exception is thrown. There are events indicating method invocation and method completion, lock acquisition and release; events generated by field loading and storing for instance fields, static fields, and arrays; events for threads starting and dying, thread switches, and `wait` and `notify` operations. In addition there are events for breakpoints and reaching a target stack depth.

Another feature of the RTI is that it allows a tool to set up a "downcall", which is a mechanism for a client program to make a direct method call into the tool. Tools register each downcall through the RTI with an owner name and a method name, for example "Tester" and "assert". This is to avoid direct dependence on class names, allowing for tool–independent downcalls to be used (enabling different implementations of tools to be used without changing every client's downcalls). Rivet's `Downcall` class contains a static method that the client can call, passing it the owner name and method name of the desired downcall and an array of arguments. For example, a programmer could insert the following code into a program before running it on Rivet:

```
Downcall.downcall("Tester",
                  "assert",
                  new Object[]{new Boolean(true)});
```

Rivet, when interpreting this method call, would invoke the `java.lang.reflect.Method` that was registered for the string pair ("Tester", "assert") and pass it as arguments the client–level array of client–level objects. The assert method would convert from client–level Boolean to lower `boolean`, and then act from there. It can return an Object that will be passed back to the client as the return value of `downcall` (so it must be a client-level object).

A.4.1 Flow of Control

The RTI gives tools the ability to invoke client methods. However, this can lead to nested interpretation, that is, having multiple copies of Rivet's interpretation loop on the lower virtual machine's stack simultaneously. This would be bad because Rivet's interpretation loop uses local variables as caches for the state of the virtual machine, which could be changed by a nested interpretation, causing the local variables in the outer interpretation to have stale values. Nested interpretation could also occur when native methods make upcalls and when the client program uses the Reflection API to invoke methods. To eliminate the nested interpretation in all of these cases, Rivet was designed to have a master loop that is external to the bytecode interpretation loop. The interpretation loop can return to the master loop (called "unwinding") in order for a new interpretation loop to begin; this suspends the client program being executed. When the new loop finishes, the old interpretation loop can be restarted at the same point it left off with no change in the state or results of the client program (unless the new loop altered the program's state). Since the loop is being reentered, the local variable caches will be refilled with the proper values.

This master loop fits in nicely with another design decision. When the Rivet Tool Interface was first created the tools were in control: the user would start up a tool which would then start Rivet. However, such a system does not allow for multiple tools to be used simultaneously. For example, our systematic tester and Eraser are both useful tools in their own right, and a user may wish to run them in parallel to ensure that a client program meets the tester's criteria. Consequently, we created a "master driver" that coordinates between the virtual machine and the tools. A user runs the master driver, indicating which tools to use and what client program to run. The driver initializes the virtual machine and then initializes the tools, giving them a chance to declare what events they are interested in. For this purpose every tool has an `init()` method with a standard signature. The driver then starts its master loop, which starts the bytecode interpretation loop that runs the client program. Whenever an event occurs that a tool has registered for, that tool's event handler is called, and when it returns the interpretation loop resumes execution of the client program. Figure A.6 illustrates the system.

When a tool wishes to invoke a client method (or perform certain other state–sensitive operations such as returning to a checkpoint) it must first request that Rivet unwind. Since Rivet must be in a consistent state to unwind, a request merely sets a flag that the interpretation loop checks on each iteration and after certain events. The tool passes a token object to Rivet with the unwind request. When the interpretation loop sees the flag, it stores its current state, exits, and returns to the master loop, carrying the token object with it. The master driver then raises an "unwound" event with the token object as an argument. This way a tool can recognize the unwind that it requested by comparing its token object to the event argument. Figure A.7 illustrates the steps of unwinding Rivet.

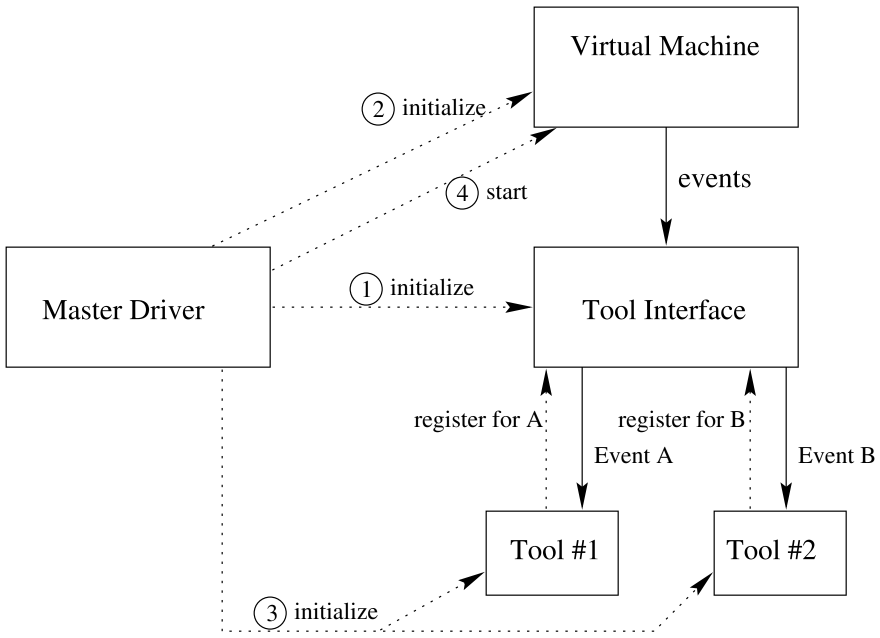

Figure A.6: Initialization of Rivet. First, the master driver creates a tool interface. Then it initializes the virtual machine. Third, the master driver initializes each tool. Finally, the master driver starts its master loop, which starts the virtual machine's interpretation loop. The interpretation loop does not return back to the master loop until an unwind is requested (see Figure A.7). The interpretation loop generates events that are sent to any tools registered for them.

A.4.2 Performance

Naturally, the power of events and checkpoints does not come for free. Table A.1 shows the results of running six benchmarks on a tool called `Events`. This tool registers for every event that the RTI makes available. It also adds several extra instance fields to a number of client classes, including `java.lang.Object`, in order to have extra fields in every client object. In addition, it was run with checkpoints enabled (as will be explained in Section A.6.2, Rivet has two modes, one in which checkpoints are disabled for efficiency and one in which they are enabled, meaning that checks are performed on field reads and writes to ensure that the proper versions of client objects are being used). The tool does not actually do anything at run time; its purpose is to measure the cost of having Rivet raise events, add extra fields, and use checkpoint–enabled fields. The performance of checkpointing by itself is discussed in Section A.6.3.

As the table shows, Rivet's features carry a penalty of ten to fifteen percent of execution time for most of the tests. The JavaCUP test's ratio is anomalous; we could

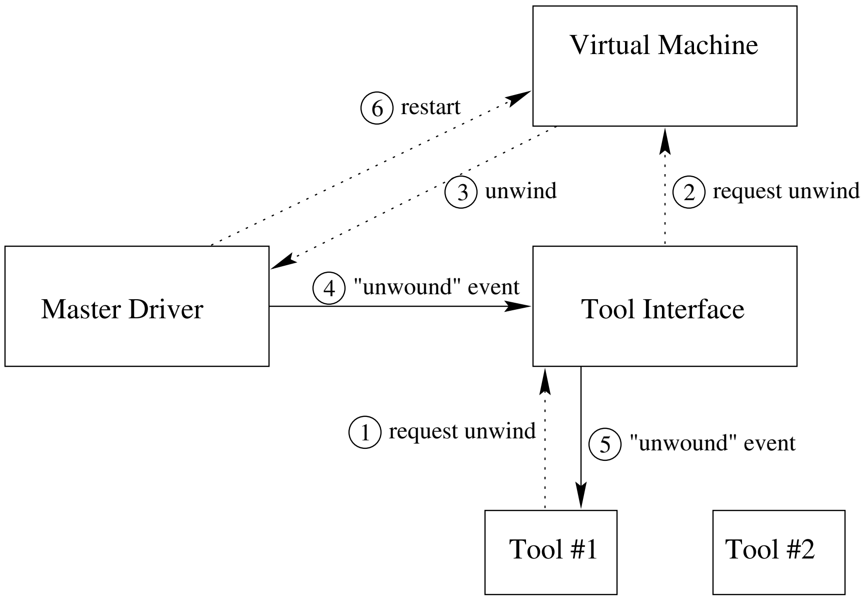

Figure A.7: How an unwind works. First, the tool requests that Rivet unwind. The tool interface forwards this to the virtual machine, whose interpretation loop returns back to the master loop. The master driver generates an "unwound" event, which the tool will receive if it registered for it. Finally, when the tool's "unwound" event handler returns, the master loop starts up the interpretation loop again.

Benchmark		No tool		`Events` tool		
	java	Rivet	slowdown	Rivet	slowdown	ratio
SPEC _201_compress	75.415	17711.271	234.9	20212.622	268.0	1.14
SPEC _202_jess	2.382	250.587	105.2	283.473	119.0	1.13
SPEC _205_raytrace	9.033	1692.260	187.3	1924.986	213.1	1.13
SPEC _209_db	0.207	30.179	145.8	33.295	160.8	1.10
SPEC _227_mtrt	8.779	1683.085	191.7	1930.597	219.9	1.14
JavaCUP	14.180	1824.480	128.7	6349.000	447.7	3.48
geometric mean			159.8		218.1	1.36

Table A.1: Time taken in seconds to run six different benchmarks on JDK1.1.5 on Linux on a Pentium II 200 MHz machine with 64MB RAM. The SPEC benchmarks are from the JVM Client98 Release 1.01 benchmark suite [SPEC]. They were each run with the `-s1` flag. These are not official SPEC results. The JavaCUP benchmark involved running JavaCUP [CUP] on the syntax of the Java language. The slowdown columns indicate the slowdown of Rivet versus java; the ratio column indicates the slowdown of using all events versus not using any events. The Rivet JIT was not enabled.

find no explanation for it. The geometric mean is a penalty of one–third of execution time; however, most tools will not register for nearly as many events as the `Events` tool.

A.5 Extra Fields

Rivet has a powerful extra field mechanism that allows tools to add any number of extra fields of any type to any client class. This enables tools to store their own data in client objects. Extra fields may not be accessed by client code because the client program is not aware of their existence. Extra fields that are public or protected and not static are inherited just like normal fields. In addition, extra fields may be added to arrays (and arrays inherit extra fields added to `java.lang.Object`). They may not be added to interfaces, however.

The virtual machine itself uses extra fields internally: an extra field is added to Object to hold synchronization information, to Thread to hold a pointer to the internal thread object used by the scheduler, to Class to hold a pointer to the corresponding RClass for reflection purposes, and another to Object to hold an object's hashcode. Also, checkpointing fields are added to every object as described in Section A.6. A tool must request that extra fields be added to a class in the tool's `init()` method, before Rivet loads any classes.

In the Generic object representation, extra fields are easy to add: the arrays that hold field values are simply lengthened. Note that GObject's array of non–primitive fields must now be of type Object instead of GObject to allow for extra fields of any type (a tool may want to store its own non–client object in an extra field).

In the Native object representation, extra fields can be added to classes that we transform in our class loader with few problems. However, since we share Object, Throwable, and all array classes with the lower virtual machine we cannot directly add fields to them. Instead we keep a hashtable for each extra field that holds the values of that field for objects that cannot have fields directly added to them. A special class that implements RepField is wrapped around the hashtable to make these fields look no different from other fields to the rest of the virtual machine.

Since subclasses of Object and Throwable are expected to inherit any extra methods of those classes, we insert the special classes SubObject and SubThrowable into the inheritance hierarchy as shown in Figure A.8. We add Object's extra fields to SubObject

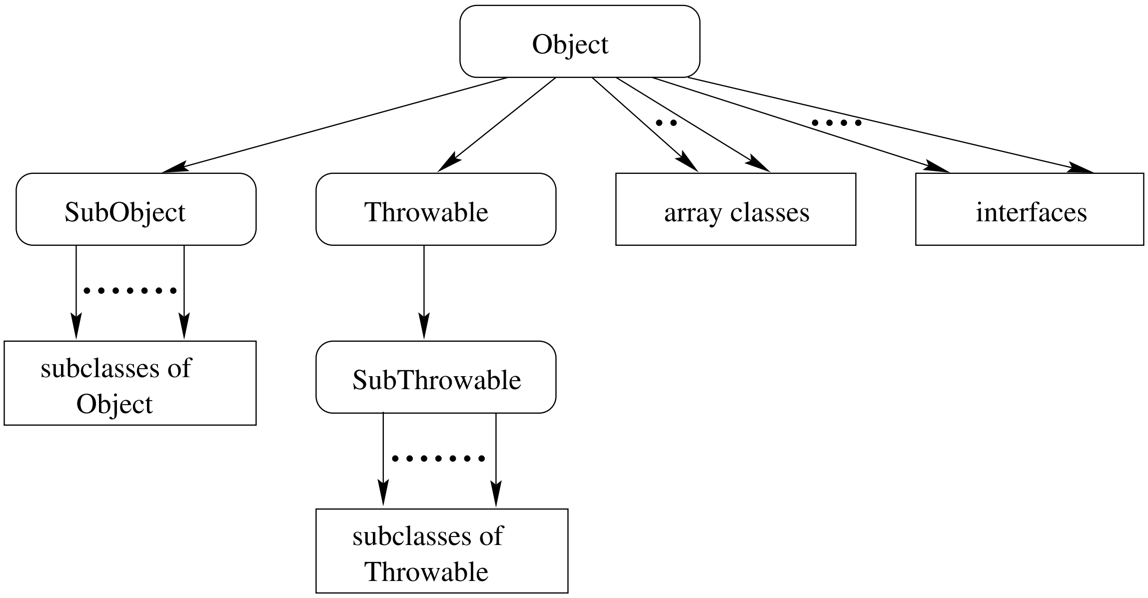

Figure A.8: Class hierarchy used for adding extra fields in the Native representation. Arrays inherit extra fields added to Object, but interfaces do not; interfaces cannot have fields at all.

and both Object's and Throwable's extra fields to SubThrowable so that their subclasses will inherit the proper fields automatically. The RepFields for these fields must first ask if a client object is really an Object or Throwable. If so, the hashtable is used; if not, the field request is forwarded to the corresponding RepField in the appropriate Sub class.

To insert SubObject and SubThrowable properly the Native class loader replaces all calls to Object constructors with calls to SubObject constructors and all calls to Throwable constructors with calls to SubThrowable constructors.

There is one problem with this implementation of extra fields for the Native representation. Since extra fields are directly inherited from parent classes through the shadow class hierarchy, the offsets of non–extra fields can change (inherited fields are placed before declared fields in instances of a class). This means that if native code uses hardcoded offsets to access fields (instead of the JNI), this code will break if it tries to access fields that have been moved. There is such code in Sun's virtual machine. We hope that future native code will always use the JNI — if so, our extra fields scheme will work. Currently we have workarounds for the native methods in the `java.io` package that use direct offsets. We have considered proposals for how to implement extra fields in a way that avoids the offset problem, but we have not yet explored these approaches.

A.6 Checkpointing

Rivet contains a powerful built–in checkpointing feature. It enables a tool to save the system state by taking a checkpoint, and at a later time return to that checkpoint. From there the program will run on a different path than the original path from that state. In order to run on the same path deterministic replay must be used (see Section A.7).

Before designing Rivet's checkpointing system we considered how it would be used. We envisioned a bidirectional debugger taking a checkpoint at regular intervals. When a user requests to go backward, the debugger returns to the checkpoint just before the target point in the program, and then uses deterministic replay to advance to the target point. Such a debugger takes many checkpoints and hence requires them to be inexpensive. Our design attempts to minimize the cost of taking a checkpoint.

When a checkpoint is taken we need to save the state of the thread scheduler, each thread's entire activation stack, which client classes have been loaded (we must run a class' static initialization method when encountering a class for the first time), and the field values for every client object and class. Saving every object's fields on every checkpoint would be very expensive. Our solution is to make copies of client objects lazily — only when an object is modified do we bother to save a separate copy of it.

We also need to save the native–level state of the client program. However, we cannot do so efficiently without requiring that native methods use library routines that we provide to allocate their memory. We have not investigated implementing this, so for now we require that client programs do not keep such native state. The core JDK has native state in the form of tables of file handles, etc., which fortunately do not cause us any problems because they do not interact visibly with Java–level client code.

Since taking checkpoints must be inexpensive, we do not want to require a tool to unwind Rivet before taking a checkpoint. For this reason we allow tools to request a checkpoint at any time; however, the only guarantee we make is that the checkpoint will be taken before the next bytecode is executed. This allows us to delay checkpoint requests made during native method execution until after the native method completes and the system is in a stable state that can be reentered. We cannot take checkpoints during native methods because we have no way of saving the lower virtual machine's stack. This means that we cannot take checkpoints during upcalls, either. This should not be a problem since upcalls are rare.

Returning to a checkpoint is a more drastic event than taking a checkpoint. It causes the virtual machine's state to change. Thus we require a tool to unwind Rivet before returning to a checkpoint.

A.6.1 Checkpointing Algorithm

The basic idea is to version objects and checkpoint them incrementally. We copy things at the object level, i.e., we do not make copies of primitive fields (we copy their containing objects). Every client object has a version that is checked on accesses to it to see if the object is stale or needs a new version made. Although we do clone client objects lazily, we clone the core virtual machine state (the threads and their activation frames) on every checkpoint.

The notion of time in the client program is captured with the "checkpoint version," which is an integer. There is a global version number called `currentCheckpoint` that indicates the current version of the client program. Each checkpoint that is made clones the virtual machine state and then increments the `currentCheckpoint`. Cloning the state involves cloning the scheduler, which clones the threads, which each clones its own activation stack. The stacks are only shallowly cloned, however; client objects referenced are not cloned at this point (remember that we are cloning objects lazily).

In addition to client program time, there is also external, real world time. This is recorded in the form of the "checkpoint count". There is a global count called `checkpointCount` which is incremented every time a checkpoint operation is performed, both when creating a new checkpoint and returning to an old checkpoint. From an external point of view, the history of checkpoints is a graph, while from the client program's point of view the history is simply a timeline. An example history graph and corresponding timeline are illustrated in Figure A.9. In this example, three checkpoints were created, followed by a return to the first checkpoint, and then two more checkpoints were created.

Every object needs to store a pointer to its "old" versions in order to return to them. Thus every object has its own timeline. Each object must also remember what count it corresponds to — counter–intuitively, the version without the count is not sufficient.

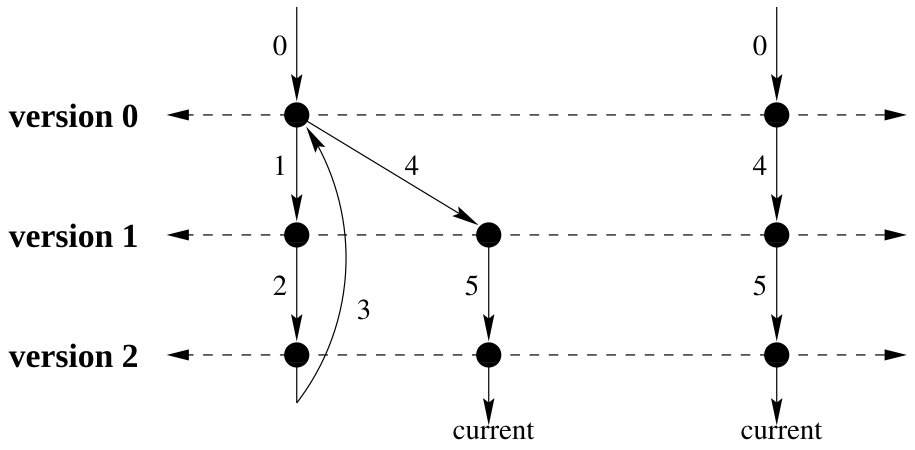

Figure A.9: An example global checkpoint history is shown on the left, and the corresponding timeline is shown on the right. The circles are checkpoints, and the numbers on the transitions correspond to checkpoint counts. The version number of a checkpoint is equal to its depth in the history graph or timeline.

Consider the objects in Figure A.10, which exist in a program whose global history is that of Figure A.9. Suppose that at the current point in the program, which is version 2 (and count 5), we wish to access object `A`. If we only knew its version number, we would have no way of knowing if it is current or not. Its version value 2 is ambiguous: it could be stale, coming from the checkpoint at count 2, or fresh, coming from the checkpoint at count 5. For this reason we must store the counts with each copy of the object. The versions could be looked up from the global history, since each count has a unique corresponding version, but for efficiency we store them with the object. The time saved is more important than the space lost because checks that examine the version happen frequently (on nearly every field load and store, as described below).

Before every object modification we detect if a checkpoint has been made since that copy of the object was created. If so, we clone that object and set the original copy to point to the clone as its previous version. On every object access, if we detect that we have gone back in time to before this version of the object existed, we follow the chain of previous versions to find the proper object; then we copy the proper object's fields into the current object. The copy step is necessary in order to keep the original object handle the same throughout. Doing so is very important, as references to this object could be anywhere and we do not want to hunt them down and update them. Thus in Figure A.9 each of object `B`'s versions points to the original client object `A` and *not* to any of its previous versions.

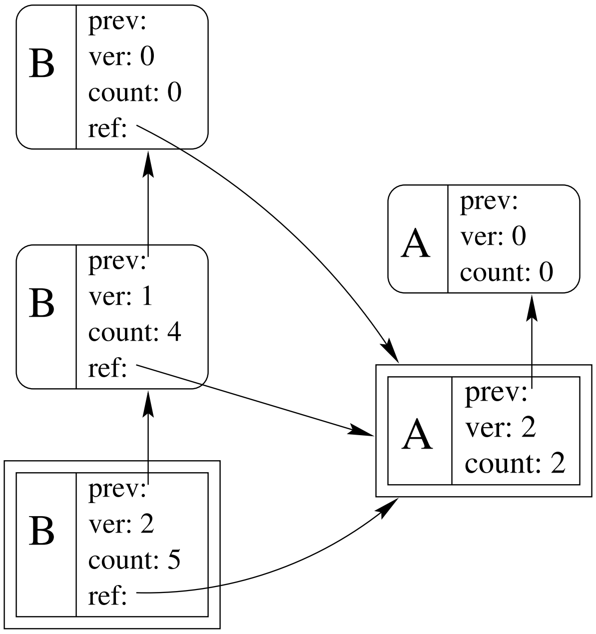

Figure A.10: Two objects in a program that has the global history shown in Figure A.9. Object `B` contains a field named `ref` that references object `A`. The current client objects are double rectangles with square edges, while clones are single rectangles with rounded edges. Note how clones are only pointed to by the previous version pointers used in checkpointing and never by client field references. The client object handles never change during execution of the program.

We have a single global history graph and a separate local history timeline for each object. On every object access we need to bring that object's local history into accordance with the global history. We need to revert to a previous version if a checkpoint with a version prior to the object's version was returned to at a count later than the object's count. Instead of storing the entire history graph, all we need is a data structure that can determine the earliest version of all checkpoint counts beyond the object's count. If the earliest version is less than the object's version then a checkpoint earlier than the object has been returned to. For efficiency we can store the count of the last return to a checkpoint and only check the earliest–version data structure when this `countAtLastReturn` is greater than the object's count.

If we are about to modify an object, we need to create a new checkpoint for that object if the global count is greater than the object's count, which means that a checkpoint has been created since the object was last checkpointed (remember that we immediately create a new checkpoint after returning to a previous checkpoint). Again we have no need

```
/** If the object passed in is stale, updates it to the
 *  current checkpoint.
 */
public void onRead(Object obj) {
  if (countAtLastReturn > obj.count)
      update(obj); // may be stale
}

/** If a new checkpoint has been created since this object was
 *  last checkpointed, makes a new version of the object
 */
public void  onWrite(Object obj) {
  if (checkpointCount > obj.count)
    newVersion(obj); // new checkpoint
}
```

Figure A.11: Checkpointing algorithm's methods to check if an object needs to be updated or if a new version of an object needs to be created.

to store the history graph.

Figure A.11 shows pseudocode for the two methods that check whether or not an object is valid, and if not perform appropriate operations to update the object. For a read the `onRead` method is used, for a write the `onWrite` method. This pseudocode assumes objects have three fields, `count`, `ver` (the version), and `prev` (points to the previous version of the object).

Before setting any field of an object, a RepField calls `onWrite` on that object. The approach for reads is more efficient. We can avoid a call to `onRead` on every field read if we ensure that all objects on the stacks and in the local variables are up–to–date. We do this by calling `onRead` on every object copied from a method caller's frame to the callee's frame, and on every object retrieved from a field (so on every read of a field of reference type). We also call `onRead` on every object in the current thread's stacks and locals whenever we return to a checkpoint. Threads store the `countAtLastSwitch`, the checkpoint count of the point where they last executed. Upon switching back to a thread, if the global count equals the stored count, we do nothing; otherwise, we call `onRead` on every object in the stack or locals of every frame on the thread's activation stack to be sure we have the right versions. Alternatively, once we have the JIT working, we can have two versions of every method: one that checks object versions on every field read and one that has no such checks. On a thread switch, we would use the former version for the first method we switch into, and use the latter everywhere else.

Figure A.12 gives pseudocode for the `update` method that reverts an object to an earlier version of itself. Since we want to keep the original client object handle, we must copy the fields from the previous version clone into the original object; this is shown in the pseudocode as the `checkpoint_copyFieldsOf` method. Its implementation will be discussed later. The `update` method has some code to deal with special cases such as interned strings. When we go back in time, for efficiency we do not bother to update the table of interned strings to what it was since the client cannot tell the difference. This means we can encounter objects that should not exist at the present moment in the client program. The `update` method simply updates the version of any such objects it encounters.

The earliest checkpoint after a given checkpoint count is computed by the `earliestCheckpointAfter` method shown in figure A.13. It uses two arrays of `int`s, one holding version numbers and the other the largest checkpoint count for which the corresponding version is the earliest. For example, for the global history of Figure A.9, the version array would be `[0,1,2]` and the count array would be `[3,4,5]`. This representation is used to save space: typically one version is the earliest for a very large number of consecutive counts.

Figure A.14 shows the method that creates a new checkpoint of an object. It must first make sure that the object is not stale. Otherwise the clone of the object could have a version and a count in the future. What it refers to as the `checkpoint_clone` method will be discussed in Section A.6.2.

A.6.2 Checkpointing Implementation

The Checkpoint class holds the checkpointing system's state and contains the code for updating objects. It has variables corresponding to those global variables discussed in the previous section: `currentCheckpoint` is the current version of the state of the client program, while `checkpointCount` is the current count. They both start at 0. Taking a checkpoint increments both of them, clones the core virtual machine state (the client threads and their activation frames), and remembers what classes have been loaded. To prevent excessive cloning of threads' frames, clones of threads that are not currently executing share the original frames. Only when one of these clones is scheduled to run do we copy the frames. This optimization reduces the amount of cloning by a significant fraction.

```
/** Update to the most recent version of this object */
public void update(Object obj) {
  // Find most recent shared point bet. global timeline & obj timeline
  int ver = obj.ver;
  // First, find earliest version in global timeline after obj's count
  int newVer = earliestCheckpointAfter(obj.count);
  if (ver > newVer) { // obj is indeed stale
    // Then, walk backward in obj timeline until hit <= earliest ver
    Object newObj = obj.prev;
    // We need the null check because we do not ever remove some types
    // of objects (like interned strings), so they may have no proper ver.
    while (newObj != null && newObj.ver > newVer) {
      newObj = newObj.prev;
    }
    if (newObj != null) {
      // Copy all the fields of newObj (we must keep "this" constant since
      // the client holds a reference to it), including ver, count, prev.
      obj.checkpoint_copyFieldsOf(newObj);
    } else {
      // No appropriate ver (obj shouldn't exist): simply update this one
      obj.ver = currentCheckpoint;
      obj.count = checkpointCount;
    }
  }
}
```

Figure A.12: Pseudocode for the method that reverts a stale object to a previous version.

As described in Section A.3, references to threads and locks are indices into master arrays instead of direct references. This level of indirection makes checkpointing much simpler. It means we do not have to change all references in the cloned state of the scheduler to point to the cloned threads instead of the original threads. Using the solution adopted for client objects, to keep the original handle always, would mean too much field copying (remember that the scheduler gets cloned *every time* a checkpoint is made).

The core virtual machine state is stored in a class called VMContext. VMContext has an internal class State that holds all of the checkpointable data (the scheduler, the threads and their frames). It is State that is cloned, and the VMContext merely points to the new State, keeping a backpointer to the old one. Checkpointing of client objects is done lazily, when objects are fetched or written to, as described in the previous section. We discuss implementing this below.

To return to a checkpoint, the Checkpoint class sets the `currentCheckpoint` to

```
/** Returns the least version whose count is >= count passed in */
private int earliestCheckpointAfter(int count) {
  // binary search for count
  int max = earliestLength;
  int min = 0;
  int idx = max / 2;
  while (true) {
    if (earliestCount[idx] < count) {
      min = idx;
      idx = (max + idx) / 2;
    } else if (idx == 0 || earliestCount[idx-1] < count) {
      break;
    } else {
      max = idx;
      idx = (idx + min) / 2;
    }
  }
  return earliestValue[idx];
}
```

Figure A.13: This method is used by the `update` method in Figure A.12. It uses two arrays, one holding versions (`earliestValue`) and the other the largest counts for which those versions are the earliest versions for the rest of time (`earliestCount`). The number of valid entries in the arrays is held in `earliestLength`.

```
/** Make a new version of this object */
public void newVersion(Object obj) {
  // Make sure we add the new object in the right place
  if (countAtLastReturn > obj.count)
    update(obj);
  // Must keep "this" constant, so make the clone the old version
  Object oldObj = obj.checkpoint_clone();
  obj.prev = oldObj;
  obj.ver = currentCheckpoint;
  obj.count = checkpointCount;
}
```

Figure A.14: Pseudocode for creating a checkpoint of an object. Note that it must make sure the object is not stale first by calling `update`.

the value desired, increments the `checkpointCount`, and notifies the VMContext. The VMContext follows its backpointers to the old state. The now future states will be garbage collected since nothing points to them anymore. Objects are updated lazily except for those in the current thread's stack and local variables, as described in the previous section. The semantics of returning to a checkpoint that we want are that the checkpoint returned to can be returned to again later. In our implementation, we need to create a new checkpoint immediately after returning in order to keep a copy of the original checkpoint returned to.

When we return to a checkpoint we need to "unload" classes so that their class initialization routines will be called again. To accomplish this, RClass has a flag saying whether that class has been "officially" loaded and another for whether it has been "officially" initialized. Officially means with respect to the client. These flags are turned on when the class if first loaded and initialized. When we make a checkpoint we store a list of all classes that were loaded since the last checkpoint (RClassLoader tells VMContext every time it loads a class). When we go back in time, as we walk back through versions, we "unload and uninitialize" (by setting the flags) every class that was loaded since the checkpoint we are heading for. The bytecode interpreter uses a special class name resolver that claims a class has not been loaded if the loaded flag is off (even if the class has been loaded by Rivet and has an RClass). When the interpreter generates a load fault for a class that has really been loaded, it zeros the class' static fields and re–executes its class initialization method to simulate re–loading it. A different class name resolver that ignores the flags is used by parts of the virtual machine that want to know if the class has really been loaded by Rivet, ignoring whether the client should know about it or not.

Checkpointing of Client Objects

The checkpointing algorithm makes use of three fields present in every object: a version `ver`, a count `count`, and a previous object pointer `prev`. We use Rivet's extra fields mechanism (see Section A.5) to add these fields to `java.lang.Object`.

Since tools adding extra fields to an object will want them to be checkpointed along with the object's normal fields, and tools will often be storing non–client objects in those fields, we have the Checkpointable class. All checkpointing is either of a client object or of a subclass of Checkpointable. Checkpointable is an abstract class that contains the three checkpointing fields and the methods `onRead`, `onWrite`, `checkpoint_clone`, and `checkpoint_copyFieldsOf`. A tool must call `onRead` or `onWrite` before every read or write to an object stored in an extra field (except for access through an RTIField, of course, which does its own version checking). Since the tools' needs and GObject's needs overlap, GObject inherits from Checkpointable. Thus we uniformly treat tool and internal virtual machine checkpointed non–client objects in the same manner.

The fact that tools must call `onRead` and `onWrite` at the appropriate times for every Checkpointable object that they store is a bit cumbersome. Another solution might be to require all non–client objects that will be checkpointed to be stored behind some interface, and all access to such objects would have version checking done automatically. There are tradeoffs between ease of use and efficiency here. Currently there is no such interface, but experience has shown that it is all too easy to leave out calls to `onRead` or `onWrite`, leading to subtle bugs in tools.

Whenever the virtual machine creates a client object it calls the `initialize` method of the Checkpoint class to set the object's version and count fields to the current global values. The access methods of the RepFields contain the `onRead` and `onWrite` method code for efficiency (rather than having them call methods in Checkpoint). The RepFields directly call Checkpoint's `update` and `newVersion` methods. For client objects, these use RepInterface methods called `cloneObjectNoCheckpoint` and `copyFields` to create new objects and update old ones. They have to use special "NoCheckpoint" methods for cloning that themselves use special object creation methods that do not initialize the checkpoint fields and do not trigger client object creation events. Also, because we use RepFields to read and write the checkpointing fields themselves, we must use special methods `setIntNoCheckpoint` for `ints` and `setNoCheckpoint` for reference fields that write to a field without performing `onWrite`. We do not bother with "NoCheckpoint" methods for reading primitive fields because we do not do checkpointing on primitive field reads as explained in the algorithm description. For Checkpointable objects Checkpoint has separate methods `updateCheckpointable` and `newVersionCheckpointable`, which call Checkpointable's `checkpoint_clone` and `checkpoint_copyFieldsOf` methods to clone and update objects.

To checkpoint array element accesses, we treat arrays like objects whose instance fields are array elements. We have special "NoCheckpoint" versions of the array copying methods for cloning arrays.

Checkpointing of static fields requires extra work in the Native representation. In the Generic representation it does not because static fields are stored in a special GObject for each class, and since GObject is a subclass of Checkpointable checkpointing of static fields works just like that for instance fields. For the Native object representation, however, static field values are kept in the lower virtual machine's class object, which we do not have direct access to. Thus we need to store the old static values separately, and we need every field read to call `onRead` because the object owning the field, the class object, is not checkpointed by us. Alternatively, we could have every return to a checkpoint go through and update every single loaded class. The efficiency of this approach relative to the cost of calling `onRead` on every field read has not been investigated.

A static field has a one–to–one relationship with its RepField, simplifying the storage problem: each RepField for a static field needs to keep track of the history of just one value. We have static fields store old values in classes called Holders, one for each type (StaticShortHolder, StaticDoubleHolder, etc.). These Holders are what is checkpointed (they each extend Checkpointable). For efficiency we do not need to update the Holder's value with the real static value on every write to a static field; we only need to update when we clone the Holder.

Extra fields stored in hashtables are checkpointed just like other fields. Static extra fields use Holders to store their fields' histories, and can use the Holder to store the current version as well without worrying about being consistent with an official value stored by the lower virtual machine.

For efficiency, we do not want RepFields to be calling checkpointing methods when the client program has no intention of ever taking any checkpoints. Thus each variety of RepField has two types: one that performs all of the checkpointing checks and one that does no checkpointing at all. Which type is created depends on a flag set via an argument to Rivet when it starts up. We end up with different versions of RepField for arrays, instance fields, static fields, extra instance fields, and extra static fields. In addition, we need to handle Throwable's (non–extra) fields specially (remember that we share Throwable with the lower virtual machine). For each of these field types we have separate classes for checkpointing versus non–checkpointing modes. The classes used to implement RepField for the various types of fields are shown for the Generic representation in Table A.2 and for the Native representation in Table A.3.

Field Type	No Checkpointing	Checkpointing
array	GArrayAccess	GCheckedArrayAccess
instance	BooleanField	CheckedBooleanField
	ByteField	CheckedByteField
	...	...
static	StaticBooleanField	StaticCheckedBooleanField
	StaticByteField	StaticCheckedByteField
	...	...

Table A.2: Classes used to implement RepField for the Generic representation for the various types of fields. These classes all checkpoint the GObject used to hold the field values (a GObject is used for static fields and array elements as well as instance fields). Since GObject is Checkpointable, all of these classes call `updateCheckpointable` or `newVersionCheckpointable`. There are different classes for each type of both instance and static fields; thus in addition to the classes for `booleans` shown in the table there are classes for `floats`, `ints`, etc.

Field Type	No Checkpointing	Checkpointing
array	NArrayAccess	NCheckedArrayAccess
instance	NField	NCheckedField
static	NField	NStaticCheckedField
extra instance	NExtraField.InstanceField	NExtraField.InstanceCheckedField
extra static	NExtraField.StaticField	NExtraField.StaticCheckedField
Throwable	NThrowableClass. NThrowableField	NThrowableClass. NThrowableCheckedField

Table A.3: Classes used to implement RepField for the Native representation for the various types of fields.

Benchmark	java	No checkpoints		Checkpoints		ratio
		Rivet	slowdown	Rivet	slowdown	
SPEC _201_compress	75.415	17711.271	234.9	18049.153	239.3	1.02
SPEC _202_jess	2.382	250.587	105.2	255.660	107.3	1.02
SPEC _205_raytrace	9.033	1692.260	187.3	1720.767	190.5	1.02
SPEC _209_db	0.207	30.179	145.8	30.498	147.3	1.01
SPEC _227_mtrt	8.779	1683.085	191.7	1721.927	196.1	1.02
JavaCUP	14.180	1824.480	128.7	1904.000	134.3	1.04
geometric mean			159.8		163.3	1.02

Table A.4: Time taken in seconds to run six different benchmarks on JDK1.1.5 on Linux on a Pentium II 200 MHz machine with 64MB RAM. The SPEC benchmarks are from the JVM Client98 Release 1.01 benchmark suite [SPEC]. They were each run with the `-s1` flag. These are not official SPEC results. The JavaCUP benchmark involved running JavaCUP [CUP] on the syntax of the Java language. The slowdown columns indicate the slowdown of Rivet versus java; the ratio column indicates the slowdown of using checkpoints versus not using checkpoints. The Rivet JIT was not enabled.

A.6.3 Performance of Checkpointing

Time Performance

In terms of efficiency, checkpointing in the Generic representation is not bad. For the Native representation, however, we have to use reflection to copy all of the fields for both `cloneNoCheckpoint` and `copyFields`. Calling checkpointing methods on every write and on object reads is a minor performance loss. In applications that do not do any actual checkpointing, the cost of using the checkpoint–enabled fields is very low. Table A.4 shows the results of running Rivet on six benchmarks, first with checkpoints disabled and then with checkpoints enabled. None of the benchmark programs take any checkpoints. As the table shows, using the checkpoint–enabled fields slows a program down by about two percent.

Time taken to make checkpoints depends on the underlying virtual machine's efficiency at memory management, as indicated by using the `Checkpoints` tool. This tool makes n checkpoints in a row and then returns to each of those n checkpoints, one after the other. The results are shown in Table A.5 for both a program with one thread and

n	Make Checkpoint		Return To Checkpoint	
	1 thread	5 threads	1 thread	5 threads
50	0.44	0.62	0.42	0.90
100	0.39	0.58	0.41	2.26
500	1.81	4.77	1.40	7.09
1000	3.68	7.06	2.10	28.45
3000	9.77	N/A	5.19	N/A
5000	16.59	N/A	7.34	N/A

Table A.5: Time taken in milliseconds to create checkpoints and to return to checkpoints. Not enough memory was available to take more than 2500 checkpoints with 5 threads. Note that returning to a checkpoint entails creating a new checkpoint; thus the lines where making takes longer than returning are misleading and must be reflecting the lower virtual machine's garbage collection times. These numbers were recorded on JDK1.1.5 on Linux on a Pentium II 200 MHz machine with 64MB RAM.

a program with five threads. The table suggests that returning to a checkpoint can be faster than creating a checkpoint; however, recall that returning to a checkpoint includes making an initial checkpoint so that the checkpoint returned to can be returned to later. This indicates that these times are misleading — the underlying virtual machine's garbage collector is coming into play here.

In long testing runs, simply taking the number of checkpoints taken and dividing it by the total time taken yields results ranging from 3.6 milliseconds to 5.3 milliseconds to make a checkpoint. This would only be valid if the tester did nothing but take checkpoints. These numbers are upper bounds since the tester does more than just make checkpoints. They provide additional evidence that the larger numbers in Table A.5 are reflecting more than just checkpointing time.

Results from working with Kaffe [Kaffe], a Java virtual machine written in C, and using `fork` to make checkpoints indicated that one fork takes about 0.5 milliseconds. Table A.5 indicates that even in Java we can do as well as that for small numbers of checkpoints that do not invoke the garbage collector. We believe that our incremental checkpointing system could do far better than forking on a high–performance virtual machine.

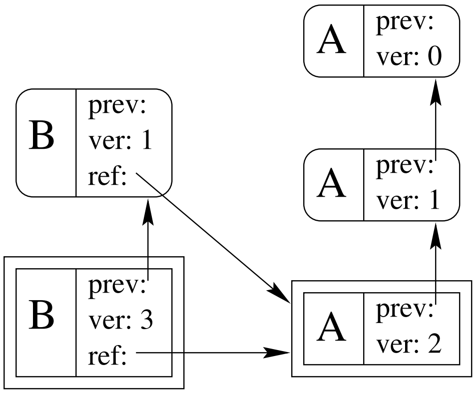

Figure A.15: Checkpointing garbage collection: picture when current version is 3. No garbage. Real client objects are represented as double rectangles with square corners; clones are indicated with single rectangles with rounded corners.

Space Performance

The old versions of objects do not clutter up memory. An original object handle points to its old version through a chain of backpointers. When we go back in time we write over the head of the chain (the `prev` pointer of the original object). That was the only pointer leading to any of the now "future" objects (recall that other client objects that refer to this one always use the original handle). Thus the unneeded and unwanted "future clones" will now be garbage collected.

Figures A.15 and A.16 illustrate a simple example of how old clones get garbage collected. As can be seen in the second figure, the old version 1 objects are now garbage since they are not reachable from anywhere.

This automatic garbage disposal of old versions does not work for objects whose `prev` field is kept in a hashtable. For these objects we have to go to quite a bit of work to garbage–collect future clones. Here are some methods for doing this that we considered and rejected:

- Give RepInterface a `returnToCheckpoint` method that is called whenever we return to a checkpoint. For the Native representation this method goes through a list of InstanceCheckedFields that exist, twice. On the first run–through it calls `markFutureClones`. `markFutureClones` goes through and calls `onRead` on every client object (not on clones of them — we must add a new extra field to every object, a flag that tells us whether an object is a client object or a clone used for checkpointing

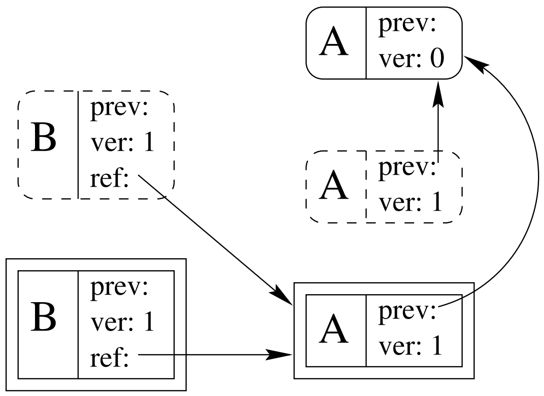

Figure A.16: Checkpointing garbage collection: picture when current version is now 1. Intermediate objects are now garbage (indicated with a dashed line). Real client objects are represented as double rectangles with square corners and are assumed to have permanent references to them so that they are not themselves garbage; clones are indicated with single rectangles with rounded corners.

purposes). Then it makes a list of all objects whose versions are in the future. On the second run–through `deleteFutureClones` is called, which removes from the hashtable everyone on the list created by `markFutureClones`. We have to do it in two passes because we do not want to delete an object that another field's hashtable contains (it might then ask for some checkpointing field and get back the default value, e.g.).

The problem with this approach is that it is terribly inefficient. We have to walk twice through a lot of hashtables on every return to a checkpoint.

- Version the hashtable used by InstanceCheckedField. This would work, but we would have to deeply–clone the hashtable, which would take too much time and memory. If we did not, the old copies of the hashtable would share objects with new copies, so those objects' state changes would be seen in the old hashtables. For example, consider arrays, whose extra fields are stored in hashtables. If an array has some of its elements changed those changes will be visible in all the old versions of itself stored in the old hashtables. If we later go back in time, the old hashtable we return to will of course contain the proper checkpointing fields for the array, indicating that it does not need to be updated. However, the array will contain the modifications made to it in the future.

What we actually do has two parts. First, we delete the future clones of an object when we do a `copyFields` on it (when we are reverting the object to an old version). The problem with this is that we never delete objects that are created and never referred to again (namely, those that do not exist in the past). To delete those objects, we do something similar to the first rejected proposal: we give RepInterface a `returnToCheckpoint` method that is called whenever we return to a checkpoint. For the Native representation that method goes through a list of all InstanceCheckedFields that exist and deletes fields for objects that should not exist at the current point in the client program. Since there are usually not too many of these (returns that do not go far back in time are more common than those that do) the `returnToCheckpoint` method only performs this expensive sweep every few hundred returns.

In the future we will want to be able to delete checkpoints. Our bidirectional debugger (based on [Boo98]) will be taking many checkpoints but wanting to keep relatively few of the older ones as time goes on. For example, it may want one checkpoint every millisecond for the last 10 milliseconds, then one every 10 milliseconds for the next 100 milliseconds, then one every 100 milliseconds for the next second, then one every second for the next 10 seconds, etc. The object checkpointing scheme allows for this with its linked–list structure: we can snip out version of objects from the middle of the chain. We would do this lazily, of course, just like the rest of the object checkpointing.

A.6.4 Limitations of Checkpointing

As mentioned earlier, we cannot currently checkpoint native state, so we require that clients do not keep any. In order to checkpoint it we would need to provide special library routines for memory allocation that tools' native method could call.

We also cannot checkpoint the lower virtual machine's stack, which means that we cannot checkpoint during native methods or upcalls.

Our incremental object checkpointing has its problems. Native methods can do field accesses without going through our checkpointing `onRead` and `onWrite` methods. If native methods access only fields on their "this" object, on which we always call `onWrite` immediately before the native call, then everything will work fine. However, we must require that clients do not access any fields other than fields of "this". In the future we hope to be able to systematically deal with JNI field access to non–"this" objects. We will always have to special–case the core JDK's direct field offset use, though.

Another limitation is that extra fields that hold reference types must be filled only with client-level objects, or with objects that implement Checkpointable. If an object not meeting these criteria is stored in an extra field, checkpointing will not work.

Finally, as mentioned in Section A.6.2, allowing tools to have extra fields be checkpointed is a very powerful feature. However, the current methods for doing so are complicated and make debugging checkpointing errors in tools difficult.

A.6.5 Alternative Checkpointing Schemes

Checkpointing schemes that we considered but discarded include:

- Add a level of indirection to avoid copying when going back in time. Around each object we would place a wrapper containing the checkpointing fields and a pointer to the current object. Then to change object versions we would just change the pointer and would not need to copy fields. The problem with this is that in the Native representation we must use the same object representation as the lower virtual machine. We cannot simply strip the wrapper off of an object before calling a native method on it because the object could refer to other objects who will still have their wrappers. This fact also kills the idea of using the Generic representation and copying fields to special lower virtual machine objects used only for native method calls.

- Have no extra fields, and keep version information in a method call's frame, with every object in there having that version. The problems are that when we make a checkpoint we will have to clone a lot of objects (all objects in the top frames of each thread), and more importantly, with an arbitrary object reference there would be no way to tell if the object is stale or not.

- Checkpoint fields, not objects. Every field could essentially be an array of values with the index into the array being the version number of that value. The problem with this is that every field in the program grows with every checkpoint made, even those that are infrequently accessed. Another choice might be to have two arrays for each field, the first holding values and the second the versions corresponding to those values. This way space is not wasted on fields that are not changing. The main problem with these field approaches is that it is very hard to delete previous checkpoints, something that we will want to do once we have tools like the debugger taking many checkpoints but wanting to keep relatively few of the older ones as time goes on. The object checkpointing scheme allows for this with its linked–list structure. Using anything other than an array for fields would lead to excessive storage overhead.

- Clone the entire system on every checkpoint. This would simply be too slow. As mentioned in Section A.6.3, doing something like a fork takes even longer than taking one checkpoint in Java. Rivet needs to be able to take many checkpoints per second. This means that it must perform incremental checkpointing.

A.7 Deterministic Replay

In order to replay deterministically we need to know when class loads happened, what input or output was performed, and when thread switches occurred (including what thread was switched to). Note that since we need to log all input and output, we must require that native methods other than those in the standard `java.io` package perform no input or output. We also need to record when `finalize` methods were called and play tricks with the garbage collector to make sure they are called at the same time again. Or we could use something like the tester's finalization criterion and assume that the user does not care about exact replay of finalizers; this is the choice we made.

The "when" for all of these logged operations is a global operation counter (it is already used for thread preemption and instruction–stepping). For every bytecode executed we increment the counter. On replay, we just wait until we reach a counter value where something should happen (like a thread switch) and make it happen then. There are three modes of execution: normal execution, record, and replay. Normal execution does not interact with the replay system. Record mode is normal execution with the addition of logging all events that need to be replayed, while replay mode replays the recorded events.

In order to record that something happened at a certain instant in time, we need to make sure we have the semantics of these instants correct. We are using an operation count, so we need to be precise about when we increment it. We do not want to increment the count, perform some operation followed by an event we want to record, and then log that the event happened at that count. On replay we would execute the event *before* performing the operation, which is incorrect. For this reason, the core interpretation loop contains the method `instructionDone`. This is called after every operation. `instructionDone` increments the operation count and, if in replay mode, checks to see if an event should be replayed. Sensitive operations like native method calls must be careful about when they call `instructionDone`. We need to call it whenever we are at a point where an event we would like to replay could occur.

`instructionDone` is called after an invocation (`invoke*` bytecode) of a non–native method and again after the return (`return*` bytecode). However, it is not called for the invocation of a native method. It is only called for a native method after the method completes. This is because to Rivet native methods are atomic and we do not want events happening in the middle of them. This means we cannot handle events in native upcalls, but we cannot handle thread switches or checkpoints or input or output there anyway.

`instructionDone` signals class load faults and class initialization faults by setting a flag in the current thread. These flags must go in the thread and not in the VMContext because during the recording execution a fault could happen right after a thread switch was asked for but before it was carried out, in which case `instructionDone` would hit it *before* the thread switch on replay if it were kept in the VMContext — we need to save it for when we reschedule that thread. These flags are checked on each iteration of the `runThreadForAWhile` loop.

When `runMethodForAWhile` returns, the opcode at the current PC has not been completed nor has instrutionDone been called on it. In some cases the PC is that of an instruction that attempted to execute but needs something else done before it can execute (load fault, init fault, monitorenter), in which case that something else will be done and the same opcode re–executed; in other cases the PC is that of an instruction that has nearly completed but just needs some higher level actions performed (invoke, return, exception), in which case those actions will be performed, `instructionDone` called, and the PC rolled onward. Thus, `instructionDone` is called after a monitorexit, but not after a monitorenter that fails to obtain its lock, because we can and should re–execute the failed monitorenter, but we should not re–execute the monitorexit.

To replay the thread ordering, on replay we disable all threads but one and keep running that thread until we reach the first recorded thread switch. Then we disable it and enable the next thread. In order to replay input and output, during record mode we intercept all native `java.io` input and output methods. We go ahead and perform the real native method, but we log what goes out or is read in. On replay we do not perform the real native method. For a read we feed the logged bytes to the program; for a write we do nothing. Once we reach the end of the replay log the file position should be where it was and a future write will put its data in the proper place.

Note that sometimes we may be replaying data that was recorded in a different session. A tool may wish to supply the logged input or output bytes, and may wish to have the writes actually performed on replay. Thus there may be several modes of replay. This is unimplemented right now. We could have one object that supplies Rivet with input or output during any execution (not just replay), with its source either the lower virtual machine (which actually performs the read or write), Rivet's replay buffer, or a tool.

To replay native methods properly we need to be able to replay their state changes. We must either require that they be functional with respect to their arguments and the Java state of the system (this means that calling the native method multiple times with the same arguments and the program in the same Java state should produce the same results, so we can simply re–execute the native call to replay it), or log the changes they make to the system. Since we already require that they only change fields of the "this" object, we would not have very much to log. However, we have not yet implemented any logging, so we assume that native methods are indeed functional. We already special–case the core JDK native methods, and this should not exclude many client Java applications, since most do not use native methods.

Note that this deterministic replay is not the type required by the tester (see Section 3.3.2).

A.8 Limitations of Rivet

To summarize the various assumptions that Rivet makes for its features to work, Rivet's requirements on client application native methods are as follows (we special–case the core JDK):

- Native methods must not construct Java objects without calling Java constructors.
- Native methods must not block. Note that Rivet could remove this requirement by having a separate thread for native method calls.
- For deterministic replay to work, native methods cannot perform any input or output.

- Rivet cannot thread switch, checkpoint, or record events for replay during Java methods called by native methods. So long as such upcalls are short this should not pose a serious problem.

- Native methods should not perform field accesses on other than the "this" object, since checkpointing needs to know of all field accesses and since deterministic replay depends on native methods being functional (for now we do not even replay changes to fields of "this").

- Native methods must not use direct hardcoded offsets for field access; they should use the Java Native Interface (JNI), since direct offsets do not work under the extra fields scheme.

- Native methods should not keep native state, since Rivet cannot checkpoint it.

As discussed in Section 3.1.2, a well–constructed program's native methods normally satisfy all but the last two of these requirements. The last requirement could be removed by providing a native library that a client's native code would use to allocate memory, registering it for native–level checkpointing. The JNI requirement should be met by newer programs; JNI is being promoted as the solution to binary compatibility problems across multiple versions of Java virtual machines. However, current JDK native libraries (such as `java.io` use direct field offsets which Rivet handles as a special case by re–implementing those libraries.

Rivet has other limitations. We have our own version of Throwable that we must distribute to users. Also, in order to perform generic lower virtual machine to client object conversion (for when native methods violate our assumptions and construct client objects without calling Java constructors) we need a core JDK `classes.zip` in which all fields are public. Thus we must distribute our own `classes.zip` or distribute a program to modify a user's existing `classes.zip`, either of which is undesirable.

Bibliography

[Boo98] Bob Boothe. "Algorithms for Bidirectional Debugging." Technical Report No.l USM/CS-98-2-23, University of Southern Maine, February 1998.

[CH98] Jong-Deok Choi and Harini Srinivasan. "Deterministic Replay of Java Multi-threaded Applications." *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pp. 48-59, Welches, Oregon, August 1998.

[CT91] Richard H. Carver and Kuo-Chung Tai. "Replay and Testing for Concurrent Programs." *IEEE Software*, 8(2):66-74, March 1991.

[CUP] CUP LALR Parser Generator for Java.
`http://www.cs.princeton.edu/~appel/modern/java/CUP/`

[Dah99] Markus Dahm, The JavaClass Application Programming Interface.
`http://www.inf.fu-berlin.de/~dahm/JavaClass/`

[FL97] Mingdong Feng and Charles E. Leiserson. "Efficient detection of determinacy races in Cilk programs." *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 1-11, Newport, Rhode Island, June 1997.

[F+96] Factor M., Farchi E., Lichtenstein Y., and Malka Y. "Testing concurrent programs: a formal evaluation of coverage criteria." *Proceedings of the Seventh Israeli Conference on Computer Systems and Software Engineering*, IEEE Comput. Soc. Press. 1996, pp. 119-26.

[GHP95] P. Godefroid, G. J. Holzmann, and D. Pirottin. "State-Space Caching Revisited." *Formal Methods in System Design*, 7(3):1-15, November 1995.

[GJS96] J. Gosling, B. Joy, G. Steele. *The Java Language Specification*, Addison–Wesley, 1996.

[God96] Patrice Godefroid. *Partial–Order Methods for the Verification of Concurrent Systems — An Approach to the State-Explosion Problem*, volume 1032 of Lecture Notes in Computer Science. Springer-Verlag, January 1996.

[God97] Patrice Godefroid. "Model checking for programming languages using Verisoft." *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pp. 174-186, Paris, France, January 1997.

[Hwa93] Gwan–Hwan Hwang. *A Systematic Parallel Testing Method for Concurrent Programs.* Master's Thesis, Institute of Computer Science and Information Engineering, National Chiao-Tung University, Taiwan, 1993.

[HTH95] Gwan-Hwan Hwang, Kuo-Chung Tai, and Ting-Lu Hunag. "Reachability Testing: An Approach to Testing Concurrent Software" *International Journal of Software Engineering and Knowledge Engineering*, Vol. 5 No.4, December 1995.

[JProbe] KL Group. JProbe Profiler.
`http://www.klgroup.com/jprobe/`

[Kaffe] Kaffe, a cleanroom, open source implementation of a Java virtual machine and class libraries.
`http://www.kaffe.org/`

[KFU97] T. Katayama, Z. Furukawa, K. Ushijima. "A Test-case Generation Method for Concurrent Programs Including Task-types." *Proceedings of the Asia Pacific Software Engineering Conference and International Computer Science Conference*, IEEE Comput. Soc. 1997, pp.485-94. Los Alamitos, CA, USA.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition*, Addison–Wesley, 1999.

[MH89] Charles E. McDowell and David P. Helmbold. "Debugging Concurrent Programs." *ACM Computing Surveys*, 21(4):593-622, December 1989.

[MD97] Jon Meyer and Troy Downing. *Java Virtual Machine*, O'Reilly, 1997.

[NW94] Robert H.B. Netzer and Mark H. Weaver. "Optimal Tracing and Incremental Re-execution for Debugging Long-Running Programs." *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pp. 313-325, June 1994.

[SPEC] Standard Performance Evaluation Corporation. "JVM Client98 Release 1.01" benchmark.
`http://www.spec.org/osg/jvm98`

[S+97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and T. Anderson. "Eraser: A dynamic data race detector for multithreaded programs." *ACM Transactions on Computer Systems*, 15(4):391-411, November 1997.

[Tai98] Antero Taivalsaari. "Implementing a Java Virtual Machine in the Java Programming Language." Sun Microsystems Laboratory Technical Report SMLI TR-98-64, March 1998.

[Tay83] R. N. Taylor. "Complexity of analyzing the synchronization structure of concurrent programs." *Acta Informatica*, vol.19, no.1, 1983, pp.57-84. West Germany.

[YT88] Michal Young and Richard N. Taylor. "Combining Static Concurrency Analysis with Symbolic Execution." *IEEE Transactions on Software Engineering*, 14(10):1499-1511, October 1988.