A Fault-Tolerant Java Virtual Machine

Jeff Napper Lorenzo Alvisi Harrick Vin
Department of Computer Sciences
The University of Texas at Austin

Abstract

The Java programming language was designed for portability and safe code distribution, not for fault-tolerance. We modify the Sun JDK1.2 to provide transparent fault-tolerance for many Java applications under the crash failure model. Our approach is to log non-deterministic events at the JVM interface using a primary-backup architecture. In particular, we identify the sources of non-determinism in the JVM due to asynchronous exceptions and multi-threaded access to shared data, as well as the non-determinism present at the native method interface. We analyze the overhead introduced in our system by each of these sources of non-determinism and compare the performance of different techniques for handling multi-threading.

1 Introduction

The Java programming language and its execution environment are designed for portability and safe code distribution. Java provides many features—such as strong typing, remote method invocations (RMI), monitors, and sandboxing—that allow programmers to develop complex distributed systems; today, Java is used in a wide variety of distributed applications, including chat servers [LLC01], web servers [Cor01], and scientific applications [WS01]. Unfortunately, the Java Runtime Environment (JRE) provides no direct support for fault-tolerance; hence, distributed applications written in Java either ignore failures, or achieve fault-tolerance through approaches—such as transactional databases or group technology [MDB01]—that are outside the scope of the JRE.

In this paper, we take a fundamentally different approach; we present the design and implementation of a *fault-tolerant Java Runtime Environment* that tolerates *fail-stop* failures [Sch84]. Our technique is based on the well-known *state machine approach* [Lam78, Sch90]. This technique involves (1) defining a deterministic state machine as the unit of replication, (2) implementing independently failing *replicas* of the state machine, (3) ensuring that all replicas start from identical states and perform the same sequence of state transitions, and (4) guaranteeing that the replication introduced to make the state machine fault-tolerant is transparent: each output-producing transition should result in a single output to the environment, rather than a collection of outputs, one for each replica.

The state machine that we choose to implement and replicate is defined by the Java Virtual Machine (JVM) specification [LY99]. The JVM is key to the portability of Java. Because the JVM is defined independently of the hardware platform that implements it, Java programs can run unmodified on any platform that implements a JVM. Hence, there are two advantages to modeling the JVM as a state machine: first, Java applications can be made fault-tolerant transparently; second, modifying JVMs implemented on different platforms allows to keep Java's "Write Once, Run Anywhere" promise even in the presence of failures.

State machines must be deterministic. Unfortunately, the JVM is not. For instance, the specification of the JVM requires support for multiple threads of execution, whose interleaving is in general non deterministic. Therefore, the same program, when run on two different JVMs with identical initial states, may cause the JVMs to go

through different sequences of state transitions, depending on the specific interleaving enforced at each JVM. We systematically identify and eliminate the effects of non-determinism within the JVM.

Managing output to the environment is a challenge in any state machine implementation. The objective is to guarantee that the output produced by a set of replicas is indistinguishable from one produced by a single state machine that never fails. Unfortunately, achieving this objective in general is impossible [Gra78], although it can be attained in special circumstances, e.g., when output actions are idempotent or when the environment can be queried to determine whether a specific output completed (*testable* output actions). Replicating the JVM's execution engine adds a new twist to this problem. The state machine does not produce output to the environment directly: instead, the execution engine invokes external procedures, called *native methods*, that are not compiled to bytecodes. Therefore, it is impossible for our state machine to recognize which output actions are idempotent or testable. To address this problem, we provide a mechanism by which native methods can be annotated so that the state machine can recognize the properties of the native methods and take appropriate action.

Our replication scheme is based on a primary-backup architecture. We use a "cold" backup, which simply logs the recovery information provided by the primary and starts processing it only if the primary fails. We implement and evaluate two techniques for eliminating the non-determinism introduced by multi-threading. The first technique allows the threads at the backup to reproduce the exact sequence of monitor acquisitions performed by the threads at the primary. The second technique replicates at the backup the thread scheduling decisions performed at the primary.

Using the original implementation of the JVM from Sun Microsystems as our performance baseline, we measure the overhead incurred by each technique in executing SPEC JVM98, a suite of representative Java applications. We find that replicating the lock acquisitions results in 100% overhead on average, while replicating thread scheduling requires only 40% overhead on average.

The rest of the paper is organized as follows. We provide background in Section 2 and an overview of the challenges involved in designing a fault-tolerant JVM in Section 3. Section 4 describes our implementation, and Section 5 evaluates the performance of our fault-tolerant JVM. We discuss the related work in Section 6, and finally, Section 7 summarizes our conclusions and presents some directions for future research.

2 Background

Java is an object-oriented programming language with support for multi-threading, mobile code, and secure code execution [GJS96]. Java programs are compiled into an architecture-independent *bytecode* instruction set. The compiled code is organized into *classfiles* containing class definitions and methods according to the Java Virtual Machine specification [LY99]. The JVM also defines standard libraries that provide supporting classes for various tasks (e.g., generic data containers, file and network I/O, and windowing components). The JVM and standard libraries comprise the Java Runtime Environment (JRE).

Java provides language-level support for multi-threading; it provides primitives for mutual exclusion (*synchronized* methods) and conditional synchronization (*wait* and *notify* methods). Threads share data objects using either thread object methods or static class data members that are shared among all instances of the class. Both the standard libraries and the order in which threads access shared data objects introduce non-determinism in the execution of programs that, as we will see, complicates the task of building a fault-tolerant JVM.

A popular approach to implementing a fault-tolerant service is to use a set of servers (called *replicas*) that fail independently. The *state machine approach* [Lam78, Sch90] is a general technique that allows to coordinate the replicas and

provide the abstraction of a single, fault-tolerant service. A state machine is a set of *state variables* and *commands*, which respectively encode and modify the machine's state. Each command reads a subset of the state variables, called the *read set*, plus, possibly, other inputs obtained from the environment; it then modifies a subset of state variables called the *write set*, and possibly produces some output to the environment. The state machine approach requires to start each replica from the same initial state and to execute at each replica an identical sequence of *deterministic commands*. A deterministic command produces the same output and write set when given the same read set (regardless of any environmental input). Under these conditions, each correct replica goes through the same sequence of state transitions and produces the same outputs.

3 The JVM as a State Machine

Modeling the JVM as a state machine raises several challenges. First, not all commands executed by a JVM are deterministic. Second, replicas of a JVM do not in general execute identical sequences of commands. Third, the read set for a given command is not guaranteed to contain identical values at all replicas. Typically state machines are used to model a single thread of execution. However, the JVM is intrinsically multi-threaded, resulting in much added complexity. Our approach to address these challenges is to renounce modeling the JVM as a single state machine: rather, we model the JVM as a set of cooperating state machines, each corresponding to one of the JVM's threads. In particular, we choose as our state machines a set of *bytecode execution engines* (BEE) inside the JVM. Though BEEs do not explicitly exist as components of the JVM, we can conceptually associate a BEE with the set of functions that perform bytecode execution and track the state of each thread. We consider the set of executing BEEs as the set of state machines comprising a replica of our fault-tolerant JVM.

The commands of the BEE state machine are bytecodes, and the state variables are the values of memory locations accessible to the BEE. Each BEE has exclusive access to its own *local state variables* and may share with other BEEs access to *shared state variables*. Our task is to ensure that each BEE replica processes the same sequence of deterministic commands. We list below the sources of non-determinism that complicate this task, and discuss how we address each of them.

3.1 Asynchronous Commands

A command is *asynchronous* if it can appear in a non-deterministic position in the sequence of commands processed by a BEE. Replicas of the same BEE may encounter an asynchronous command at different points in their command sequences. In some state machines (e.g., [BS95]), asynchronous commands correspond to hardware interrupts. Although there are interrupts in the JVM, they do not correspond to asynchronous commands. For example, the JVM performs I/O synchronously, and any I/O completion interrupt that corresponds to a given bytecode is delivered before the execution of that bytecode completes.

In the JVM, asynchronous commands correspond to asynchronous Java exceptions that are not interesting sources of non-determinism. All but one of these exceptions are raised by fatal errors in the run-time environment (e.g., resource exhaustion) or in the implementation of the JVM (e.g., locks in inconsistent states). Such errors are intrinsic to the run-time environment of the application and would repeat themselves if all replica environments were identical. Our implementation must take care not to replicate these exceptions because replication would obviously cause all replicas to fail. We assume that either such errors do not occur or that the replicas' run-time environments are sufficiently different:

R0 Environment and JVM implementation exceptions are not raised at all replicas.

The standout non-fatal asynchronous exception is delivered to a thread when it is killed by another thread. However, use of this exception

is deprecated beginning with the Java Development Kit version 1.2. We therefore place the following restriction upon applications:

R1 A thread may not invoke the `java.lang.Thread.stop` method.

3.2 Non-deterministic Commands

A command is *non-deterministic* if its write set or its output to the environment are not uniquely determined by its read set. The only non-deterministic bytecode executed by the JVM invokes a *native method*. Java includes the Java Native Interface (JNI) [Lia99] to invoke methods that execute platform-specific code written in languages other than Java. Native methods have direct access to the underlying operating system and other libraries. By accessing the operating system, for instance, native methods implement windowing components, file and network I/O, and read the hardware clock.

Native methods therefore, in addition to the read set, may take as input values from the environment. It is in general impossible to have the replicas agree on these input values, since input is performed outside the control of the JVM. Instead, we make sure that differences in input values (e.g., different clock values) do not result in different write sets for the command. In the conventional state machine approach, replicas run an agreement protocol to make their write sets identical. In our case, this protocol simply forces the backup to adopt the write set produced by the primary. However, since native methods execute beyond the purview of the JVM, an agreement protocol cannot ensure that replicas executing a native method will behave identically. We thus restrict the behavior of native methods as follows to achieve identical results at all replicas:

R2 Native methods must not produce non-deterministic output to the environment.

R3 Native methods must not non-deterministically invoke other methods.

R2 restricts the native method behavior visible to the environment; however, it is often possible to circumvent this restriction and still obtain the same functionality provided by the offending method. For example, a method that reads the current time and then prints it could be split into two methods. The first method reads in the local time and writes it to some local variable *lc*, which constitutes the method's write set. Our agreement protocol will ensure that executing the first method at the primary and the backup results in the same value for *lc*. The second method, which prints the value of *lc*, now produces deterministic output to the environment.

R3 restricts the ways in which a native method may invoke other methods. While executing outside of the state machine, a native method can invoke Java methods, causing the BEE to execute commands. If a native method calls a Java method non-deterministically (e.g., if the native method decides to acquire a lock depending on the value of the local clock) the sequence of commands processed by a BEE may be different at each replica. We rule out this possibility by forbidding native methods from making non-deterministic calls to Java methods. We do not consider R3 to be a restriction, but rather a better programming paradigm: to avoid debugging nightmares, it is wise to restrict non-determinism in native methods to input methods.

3.3 Non-deterministic Read Sets

Multi-threaded access to data creates the possibility of deterministic commands reading different read sets at different replicas of a given BEE. We call a read-set *non-deterministic* if it contains at least one shared variable.

Java allows data to be shared both explicitly, by invoking methods on a shared object, and implicitly, through static data references. In general, the bookkeeping necessary to determine which objects are actually shared can result in a significant source of overhead.

One way to make this problem more manageable is to assume that every access to a shared variable is protected by a monitor:

R4A All access to shared data is wrapped by correct use of monitors (i.e., Java's

```
1  class Example {
     // Accessible from all threads.
2    static Formatter shared_data = null;

3    String toString() {
       // Guard is not protected by a
       // monitor, resulting in data race.
4      if(null == shared_data) {
5        shared_data = new Formatter();
6        synchronized_method();
7      }
8    }
9  }
```

Figure 1: A common data race in Java. If the constructor and synchronized_method are idempotent the data race has no semantic effect.

`synchronized` keyword).

A Java monitor guarantees exclusive access to shared variables: in practice, the monitor allows the invoking BEE to transform temporarily a shared variable into a local variable. To a BEE that invokes a monitor and acquires its associated lock, however, the values stored in these temporary local variables appear to be non-deterministic, since they have been last modified by some arbitrary BEE.

One way to eliminate this non-determinism would be for the replicas to agree on the *values* of the variables associated with every lock they acquire. This approach is hard to implement, however, because Java does not express or enforce the association between a lock and the variables it protects, leaving this responsibility to the programmer.

Our first approach is instead to achieve agreement on the *sequence* of BEEs that acquire each lock. Reaching agreement on a lock acquisition sequence ensures that the corresponding BEEs at the primary and the backup access the variables associated with the lock in identical order, thereby guaranteeing that all commands executed by corresponding BEEs have identical read sets.

Unfortunately, many real programs do not satisfy R4A: even the JRE provided by Sun does not correctly acquire locks on all shared data. In particular, static data members are often shared between threads without explicit shared method invocations. As we try to reach agreement on the sequence of lock acquisitions, these race conditions may cause the state of the primary and the backup to diverge, even when they do not affect the semantics of the program in which they appear. Figure 1 shows an improper use of static data members. Object shared_data, a static data member, is shared by all Example objects. Because the guard on line 4 is not protected by a monitor, different thread schedules at the primary and the backup may result in a different number of invocations of synchronized_method, preventing agreement on the sequence of lock acquisitions. Indeed, to test our implementation of replicated lock acquisitions we had to find and remove these race conditions in the JRE by hand! Though code given in Figure 1 is technically incorrect, we wanted to find a less labor-intensive way to handle this common (mal)practice.

Our second approach does not rely on R4A, but instead eliminates non-deterministic read sets by replicating at the backup the order in which threads acquire the scheduling lock at the primary. This approach requires a run-time environment that enforces the following restriction:

R4B Acquiring the scheduler lock ensures exclusive access to all state variables.

When R4B holds (e.g on a uniprocessor), a BEE that acquires the scheduler lock effectively changes *all* its shared variables to local variables, because no other BEE is allowed to execute commands. By replicating the scheduling of threads, this implementation correctly replicates access to all shared data even when locks are incorrectly used by the programmer (the JVM automatically and correctly uses the scheduling lock).

3.4 Output to the Environment

The state machine approach strives to hide replication from the environment by requiring the output to the environment to be indistinguishable from what a single correct state machine would produce. To meet this requirement, we distinguish between output to the environment that affects *volatile* state (i.e., state that does not survive failure of the state machine) and *stable*

state (i.e., state that does). A particular command can produce multiple outputs to the environment, each of which is either *volatile* or *stable* depending upon the affected state.

Hiding replication of output that modifies stable state is easy if the output is either *idempotent* or *testable*. In the former case, the output is independent of the number of times the corresponding command is executed, while in the latter the environment can be tested to ascertain whether the output occurred prior to failure. Except for these cases, it is impossible to maintain the "single correct machine" abstraction in the presence of failures [Gra78]. For instance, in a primary-backup system a backup cannot in general determine whether the primary failed before or after performing an output command and executing the command again could produce different results. We therefore introduce a further restriction:

R5 All native method output to the environment is either *idempotent* or *testable*.

Ensuring replication of volatile output may be necessary for correct operation. For example, the OS underneath the JVM is considered part of the environment of our system. Opening a file at the primary creates OS state that disappears when the primary fails and that the backup must replicate if it is to execute correctly. Some volatile state may be restored simply by replaying the output (i.e., if the methods are idempotent), but in general volatile state may require special treatment. For instance, replaying messages on a socket will not recover the state at the backup because sending messages in general is not an idempotent operation. In fact, an extra layer must be added to make sending messages either an idempotent or testable operation.

Our protocol uses a special interface, called *side effect handlers*, to replicate the lost volatile state of the primary. Native methods may create volatile state as an effect of producing output to the environment. Using JNI, any application may call native methods supplied by the application. Our interface allows an application programmer to include methods to replicate the volatile state of the primary created by the additional native methods. For example, we have included through the interface methods to handle file I/O in the standard JRE libraries. We require applications to use this interface whenever they invoke a native method that creates volatile state in the environment, leading to our last restriction:

R6 If a native method produces volatile state in the environment, a side effect handler is provided to recover the state.

4 Implementation

Our fault-tolerant JVM is based on the JVM from the Solaris Java Development Kit (JDK) 1.2 community source release. Sun's JVM provides Just-In-Time (JIT) compiling of bytecodes and two implementations of multi-threading. The *native threads* version provides thread scheduling in the underlying OS, while the *green threads* version implements multi-threading inside the JVM. To maximize portability, our implementation modifies the green threads version.

To implement primary-backup, we add two system threads to the JVM. One performs failure detection to allow the backup to initiate recovery, and the other is responsible for sending or receiving logging information at the primary or backup, respectively. These additional threads join the several pre-existing system threads that perform tasks such as garbage collection and finalizing objects. We now discuss how our implementation addresses the challenges (non-deterministic commands, non-deterministic read sets, and output to the environment) that we identified in Section 3.

4.1 Nondeterministic Commands

We checked by direct inspection and categorized all native methods in the standard libraries of the JRE: fewer that 100 native methods are non-deterministic. We store the *signature* of these methods (i.e., their class name, method name, and argument types) in a hash table. Every time a native method is invoked at the primary,

its signature is checked against those stored in the hash table. If there is a match, then the method's return values (including arguments, if they are modified) and the exceptions it raises are sent to the backup, which keeps an identical hash table. If during recovery the backup is about to execute a method whose signature is stored in its hash table, then the backup uses the return values and exceptions provided by the primary. Note that the backup may still elect to invoke the method, in order to reproduce volatile output. However, in this case the return values and exceptions generated at the backup are ignored in favor of those logged by the primary.

4.2 Nondeterministic Read Sets

Data races and differences in scheduling among the JVM's threads can make read sets that contain shared variables return different values at the primary and the backup. We use two different approaches to make read sets deterministic.

Replicated Lock Synchronization The first approach relies on the assumption that all shared data is protected by locks that, if correctly acquired and released, ensure mutual exclusion. Under this assumption, we create a mechanism that guarantees that threads acquire locks in the same order at the primary and at the backup.

Replicating the order in which threads acquire locks requires identifying the locking thread, the lock, and the relative order of each lock acquisition. We store this information in a *lock acquisition record*, which is a tuple of the form (*t_id, t_asn, l_id, l_asn*) where:

t_id is the *thread id* of the locking thread.

t_asn is the *thread acquire sequence number*. The value of *t_asn* records the number of locks acquired so far by thread *t_id*.

l_id is the *lock id*.

l_asn is the *lock acquire sequence number*. The value of *l_asn* records the number of times lock *l_id* has been acquired so far.

These records are created by the primary, but are used during recovery by the backup. Therefore, for each thread and lock, the primary needs to generate virtual *t_id*s and *l_id*s that are unambiguous across replicas. For instance, although in the JVM each lock is uniquely associated with an object, the primary cannot simply use the object's address as the lock's *l_id*, because this address is meaningless at the backup. Further, any scheme that assigns ids according to the order in which events—such as thread and object creation—occur at the primary is dangerous, since these events may be scheduled differently at the primary and the backup.

We then define recursively the id of a thread t as consisting of two values: i) the id of the parent thread of t (the parent of the first thread has by convention $t\_id = 0$) and ii) an integer that represents the relative order in which t is created with respect to its siblings. This definition is well founded because, although the absolute order in which t is created does depend on the order in which threads are scheduled, t's parent spawns its descendants in the same relative order at the primary and the backup, independent of scheduling.

To assign a lock its *l_id*, we observe that threads execute deterministic programs. Hence, the sequence of locks acquired by a thread with a given virtual *t_id* is identical at the primary and the backup. We can then uniquely identify a lock by specifying the *t_id* and the *t_asn* of the first thread that acquires the lock at the primary. We get an even simpler *l_id* as follows. When the primary acquires a lock for the first time, it assigns to the lock a locally unique value (our *l_id* is simply an integer); it then creates an *id map*, which is a tuple of the form (*l_id, t_id, t_asn*) that associates the *l_id* with the appropriate *t_id* and *t_asn*. Each map is then logged at the backup.

During failure-free execution, whenever the primary acquires lock *l_id*, it generates a corresponding lock acquisition record, and logs it at the backup. If the primary fails, the backup's threads use the logged id maps and acquisition records to reproduce the sequence of lock acquisitions performed by the corresponding threads

at the primary.

When a backup thread t tries to acquire a lock with id l, it checks if the log contains a lock acquisition record with $t\_id = t$ and $l\_id = l$, and t_asn equal to the current value of t's acquire sequence number. If such a record r exists, t then waits for its turn for acquiring lock l—that is, t waits until l's acquire sequence number is equal to the value of l_asn stored in r, acquires the lock, and removes r from the log. If the log contains no such record, then t waits until the log contains no more lock acquisition records (indicating the end of recovery at the backup) before it proceeds with acquiring lock l.

The case in which a backup thread t attempts to acquire a lock that still has no l_id requires special treatment. First, t checks if it is its responsibility to assign the id to the lock. The thread looks for an id map with $t\_id = t$ and matching t_asn; a match implies that, before the primary failed, thread t at the primary assigned to that lock the l_id stored in the id map. If a match is found, the corresponding map is removed from the log and the id of the lock is set to l_id.

If a match is not found, then either (i) the lock was assigned its l_id at the primary by a different thread t', or (ii) no primary thread logged an id map for the lock before the primary failed. Thread t handles these two cases by waiting, respectively, until either t' assigns the l_id at the backup or until the log contains no more maps, in which case t can safely assign a new l_id to the lock.

Replicated Thread Scheduling. The second approach relies on the assumption that the scheduling lock protects all shared data. We modified the green-threads version of the JVM that implements a user-level thread library for a uniprocessor, where only one thread executes at a time and all accessible data is implicitly local during its execution.

Whenever the primary interrupts the execution of a thread t to schedule a new thread, it creates a *thread scheduling record* (*br_cnt, pc_off, mon_cnt, l_asn, t_id*), where:

br_cnt is a count of the control flow changes (e.g., branches, jumps, and method invocations) executed by t.

pc_off is the bytecode offset of the PC within the method currently executed by t.

mon_cnt is a count of the monitor acquisitions and releases performed by t.

l_asn is defined when t is rescheduled while waiting on a lock and records the lock's acquisition sequence number.

t_id is the thread id of the next scheduled thread.

These records are logged at the backup, which uses them during recovery to enforce the primary's schedule.

The basic scheme for tracking how much Java code t executed before being rescheduled is simple and it is implemented by the first two entries in the schedule record. Rather than counting the number of bytecodes, which would add overhead to every instruction, we instrumented the JVM to increment *br_cnt* for each branch, jump, and method invocation [SE98]. Further, since the program counter address is meaningless across replicas, we store in *pc_off* the last bytecode executed by t as an offset within the last method executed by t. Unfortunately, in our implementation this requires an update to the thread object after executing every bytecode because it is hard to determine, when t is rescheduled, where the JVM is storing its program counter, whose value is needed to calculate *pc_off*.

A first complication over this simple scheme arises when t is rescheduled while executing a native method. Native methods are opaque to the JVM: we have no way of determining precisely when t is rescheduled. Often this is not a problem: when repeating t's schedule during recovery, the backup reschedules t right before the native method is invoked. This is unacceptable, however, if t, while executing within the native method, acquires one or more locks: reproducing the lock acquisition sequence is necessary for correct recovery, because it is this sequence that

determines the value of shared variables. Fortunately, whenever a lock is acquired or released, control is transferred back inside the JVM. Our implementation intercepts all such events, independent of their origin, allowing us to correctly update the value stored in *mon_cnt*. In this case, instead of rescheduling t during recovery before invoking the native method, we allow t to execute within the native method until it performs the number of lock acquisitions stored by the primary in *mon_cnt*.

Further complications come from the interaction of application threads and system threads. System threads do not correspond to a BEE executing application code, and several do not execute Java code at all (e.g., the garbage collector). As was the case for native threads, we cannot reproduce scheduling events that involve system threads [1]. Ignoring system thread scheduling creates problems when application and system threads share resources, such as the heap, because both types of threads may contend for the same locks.

In particular, interaction with system threads may result in the following two events occurring during the recovery of an application thread t:

t is forced to wait at the backup for a lock that was acquired without contention at the primary. In this case, t runs the risk of being rescheduled by the backup before it can complete the sequence of instructions executed by its counterpart at the primary. We solve this problem by adding a separate *scheduler thread* and a private runnable queue (as in user-level thread libraries) to guarantee that t will continue to be scheduled, without being interleaved with other application threads, until necessary.

t acquires without contention at the backup a lock for which it was forced to wait at the primary. So, while t was rescheduled at the primary, it may not be rescheduled at the backup. It is easy to use *mon_cnt* to enforce the correct scheduling.

Threads may also perform *wait* operations on a monitor, blocking the thread until a corresponding *notify* or *notifyAll* is performed. If multiple threads are awakened, we need to guarantee that they will acquire the monitor in the same order at the primary and the backup. To do so, we store the *l_asn* of the monitor lock as part of the thread scheduling record.

A final subtle point arises when the backup completes recovery, i.e. when it finishes processing the sequence of thread scheduling records logged by the primary before failing. The last scheduling record in this sequence contains the *t_id* t' of the next thread that the primary intended to schedule—the primary failed before recording at the backup the scheduling record for t'. Nevertheless, the backup must schedule t' because at the primary t' may have interacted with the environment. t' will execute at the backup until these interactions are reproduced.

4.3 Garbage Collection

Garbage collection in Sun's JVM is both asynchronous and synchronous. Any thread may synchronously collect garbage by invoking a Java system library native method. Asynchronous garbage collection is performed periodically by a separate collector thread and during memory allocation when memory pressure indicates collection is needed. Since garbage is by definition unused memory, we initially thought that we could safely avoid replicating the behavior of the asynchronous collector thread. Surprisingly, however, asynchronous garbage collection can be a source of nondeterministic read sets. Indeed, both *soft references* and *finalizer methods* create paths for nondeterministic input to application threads.

Soft references are references to objects used to implement caches. By fudging the definition of garbage, the references are guaranteed to be garbage collected before an out-of-memory error is returned to the application. Because R0 prevents such an error from being raised at all repli-

[1] Replicating thread scheduling at the OS level in the native threads library would allow us to handle all threads, but at the cost of reduced portability. Further, we would still have to modify the JVM to handle other sources of non-determinism.

cas, collection of soft references may occur at different times at different replicas. For instance, the primary may find an object in its cache, while the backup may not, leading the execution of primary and backup to diverge [2]. Although we could replicate the behavior of the asynchronous garbage collector by recording when it locks the heap, our current implementation uses a simpler solution: we just treat all soft references as strong references, which represent active objects and are therefore never collected. This shortcut has no effect on our experiments because they never created enough memory pressure to dictate the collection of soft references.

Another possible source of nondeterminism is improper use of finalizer methods. These methods are intended to allow objects to free memory that cannot be freed automatically by the garbage collector (e.g., if memory was allocated in a native method). The Java language specification states that finalizer methods are invoked on objects before the memory allocated to the object is reused, but does not specify exactly when, leaving open the possibility of different behaviors at the primary and the backup. Our current implementation assumes that finalizer methods only free unused memory: hence, since no data is shared between the thread that runs the finalizer on dead objects and any threads that previously used those objects, no new source of non-determinism is introduced. However, it is possible to write improper finalizer methods that do more than free unused memory: in fact, they may perform arbitrary actions, possibly with non-deterministic side effects. Although we don't currently replicate the invocation of finalizers, it would be easy to do so using one of the approaches discussed in Section 4.2.

4.4 Environment Output

We have developed a generic approach to output commands in native methods that we call *side effect handlers* (SE handlers). The SE handlers are used to store and recover volatile state of the environment and to ensure *exactly-once* semantics for output commands. A handler consists of five separate methods that are called at various stages of execution at each replica.

register This method is used to register with the JVM information about the native methods that the handler will manage, including the signature of the method, whether the method is a nondeterministic command and/or an output command, and whether its arguments should be logged (i.e., if they are also output arguments).

test This method is called at the backup to test during recovery whether an output command succeeded. For example, the first output command after recovery is terminated is *uncertain*—we cannot in general decide whether the command has completed [Gra78]. *test* is called on an uncertain command to determine whether a *testable* output completed before failure, guaranteeing exactly-once semantics. Commands for which the *test* method is not defined are considered idempotent and are simply replayed.

log This method is called at the primary after executing an output command. The system provides *log* with the arguments to the native method that performed the output (including the class instance object), the return value from the native method, and extra information about the internal state of the JVM. *log* saves and returns in a message all state necessary to recover the output of the command. For example, on a file write this message may store the file descriptor and the amount written (or the current file pointer offset).

receive This method is called at the backup to receive the state stored by the primary through the *log* method. Before saving the state, *receive* may compress it: for example, *receive* could compress the results of several file writes into one offset for the file pointer.

restore This method is called at the backup during recovery. It is invoked only once.

[2] Similar arguments also apply to *weak references* [], which we treat similarly.