A Fault-Tolerant Java Virtual Machine

Jeff Napper, Lorenzo Alvisi, Harrick Vin*

Abstract

We modify the Sun JDK1.2 to provide transparent fault-tolerance for many Java applications using a primary-backup architecture. We identify the sources of non-determinism in the JVM (including asynchronous exceptions, multi-threaded access to shared data, and the non-determinism present at the native method interface) and guarantee that primary and backup handle them identically. 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, web servers, and scientific applications. 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—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. Our technique is based on the well-known *state machine approach* [1, 2]. This approach 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 the replication 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.

Our approach is inspired by, and extends, the work of Bressoud and Schneider on *Hypervisor-based fault-tolerance* [3], which presents a strong case for achieving transparent fault tolerance by 1) building a software layer (the hypervisor) that implements a virtual state machine over the underlying hardware and 2) implementing replica coordination in the hypervisor. To demonstrate their approach, Bressoud and Schneider had to build an hypervisor for (a subset of) the HP PA-RISC architecture. The observation that led us to begin this work is that Java's virtual machine is already specified and implemented—a fact we leverage to simplify our task.

The state machine that we implement and replicate is defined by the Java Virtual Machine (JVM) Specification [4]. 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. Replicating the JVM state machine allows Java applications to be made fault-tolerant transparently. Modifying JVMs implemented on different platforms allows us to keep Java's "Write Once, Run Anywhere" promise, even in the presence of failures.

State machines must be deterministic for replication to work. Unfortunately, the JVM is not deterministic. We must therefore systematically identify and eliminate the effects of non-determinism within the JVM. In doing so, we face the same issues (asynchronous exceptions, output to the environment, etc.) identified in [3]. In addition, however, we must address a new challenge: multi-threading.

The specification of the JVM requires support for multiple threads whose interleaving is, in general, non-deterministic. Therefore, the same program, when run on two different JVMs with identical initial states, might cause different JVMs to make different sequences of state transitions, depending on the specific interleaving enforced at each JVM. We implement and evaluate two techniques for eliminating the non-determinism introduced by multi-threading. The first technique forces each replica to perform the same sequence of monitor acquisitions; the second technique guarantees the same sequence of thread scheduling decisions. In implementing these techniques, as well as the others used to eliminate non-determinism in the JVM, we modify mostly platform-independent code—we make platform dependent changes only to optimize performance. Although this paper reports our experience with the Sun JDK 1.2 community source release JVM running on the SPARC, the techniques proposed are broadly applicable to other platform and other JVM implementations.

Instantiating replica coordination for the JVM also gives the traditional challenges of replica coordination a new, distinct flavor. Consider the problem of producing output to the environment. The objective is to guarantee that the output caused by a set of replicas is indistinguishable from that produced by a single state machine that never fails. Achieving this objective in general is impossible, although it can be attained in special circumstances, e.g., when output actions

*Authors' address: Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712. email: jmn,lorenzo,vin@cs.utexas.edu. This work was supported in part by a grant from Sandia National Laboratories and by the Texas Advanced Research Program. Alvisi was also supported by the National Science Foundation (CAREER award CCR-9734185), an Alfred P. Sloan Fellowship, and the AFRL/Cornell Information Assurance Institute.

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*. Therefore, it is impossible for our JVM state machine to recognize which output actions are idempotent or testable. We provide a mechanism by which native methods can be annotated so that the state machine can recognize the properties of 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. Keeping the backup updated would require only minor modifications to our implementation. 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 140% overhead on average, while replicating thread scheduling incurs 60% overhead on average.

2. Background

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 [4]. The JVM also defines standard libraries that provide supporting classes for various tasks (e.g., data containers, I/O, and windowing components). The JVM and standard libraries comprise the Java Runtime Environment (JRE). Java provides language-level support for multi-threading, mutual exclusion (synchronized methods) and conditional synchronization (wait and notify methods). Threads share data objects using either methods on shared objects or static class data members that are shared among all instances of a class.

A state machine is a set of *state variables* and *commands*, which respectively encode and modify the machine's state. A 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. For a given command, the read and write sets are fixed. However, the values that these variables assume at each invocation of the command can change. Henceforth, we refer to these values as *read-set values* and *write-set values*. The state machine approach requires each replica be started from the same initial state and each replica to execute an identical sequence of deterministic commands. A deterministic command produces the same output and write-set values when given the same read-set values. Under these conditions, each correct replica undergoes the same sequence of state transitions and produces the same outputs.

Table 1: Restrictions placed on applications and execution environment.

R0: Fatal environment and JVM implementation exceptions are not raised at all replicas.

R1: A thread must not invoke `java.lang.Thread.stop`.

R2: Native methods must produce only deterministic output to the environment.

R3: Native methods must invoke other methods deterministically.

R4A: All access to shared data is protected by a monitor (i.e., Java's `synchronized` keyword).

R4B: A thread has exclusive access to all shared variables while scheduled.

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

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

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. State machines typically model a single thread of execution [3, 2] while the JVM is intrinsically multi-threaded, complicating replica coordination significantly. Our approach to address these challenges instead models the JVM as a set of cooperating state machines, each corresponding an application thread. In particular, we choose as our state machines a set of *bytecode execution engines* (BEE) inside the JVM. Although 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. The set of executing BEEs comprises the set of state machines that together define a replica of our fault-tolerant JVM. We assume that each BEE begins in an identical initial state.

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. Below we list 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 anywhere in the sequence of commands processed by a BEE. Replicas of the same BEE might encounter a given asynchronous command at different points in their command sequences. In [3] hardware interrupts are asynchronous commands. Although there are interrupts in the JVM, they do not give rise to asynchronous commands. For example, our 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. Programmers can use Java's multi-threading to perform asynchronous I/O or events.

Asynchronous commands in the JVM 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 not replicate these exceptions, or all replicas will deterministically fail. Replication is effective only if we assume that either such errors never occur or that the replicas' run-time environments are sufficiently different. We assume the latter in R0 in Table 1.

The stand-out non-fatal asynchronous exception is delivered to a thread when it is killed by another thread. However, beginning with the Java Development Kit version 1.2, use of this exception is deprecated. Applications that use this method might not work on future releases of the JVM and should be rewritten using condition variables. We therefore make restriction R1 in Table 1 upon applications prohibiting the use of the deprecated exception.

3.2. Non-deterministic Commands

A command is *non-deterministic* if its write-set values or its output to the environment are not uniquely determined by its read-set values. The only non-deterministic bytecode executed by the JVM invokes a *native method*. Java includes the Java Native Interface (JNI) [5] 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, I/O, and read the hardware clock.

Native methods therefore may take input values from the environment as well as from the read set. In the conventional state machine approach, replicas run an agreement protocol to make their read sets and the input from the environment identical. It is generally impossible to have the BEEs agree on input values from the environment, since input is performed outside the control of the JVM. Instead, we make sure that differences in input values (e.g., different local clock values) do not result in different write-set values for the command. In our case, this protocol simply forces the backup to adopt the write-set values 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 must restrict the behavior of native methods by R2 and R3 in Table 1 to achieve identical results at all replicas.

R2 restricts the native method behavior visible to the environment; however, it is often possible to relax 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 the local time and writes it to a local variable *lc*, which constitutes the method's write set. Our agreement protocol ensures 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.

```
1  class Example {
     // Accessible from all threads.
2    static Formatter shared_data = null;
3    String toString() {
       // Guard not protected by monitor!
4      if(null == shared_data) {
5        shared_data = new Formatter();
6        synchronized_method();
         // code continues...
```

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

R3 restricts the ways in which a native method invokes 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) then the sequence of commands processed by a BEE could 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 a significant restriction, but rather a better programming paradigm: to avoid debugging nightmares, it is wise to restrict non-determinism in native methods to input methods. Just as R2, R3 might be upheld by splitting an offending method into a non-deterministic input method and a deterministic method. For instance, the clock example would be handled by placing the clock read in a different method and allowing our replicas to agree on the local clock values before invoking the (now deterministic) method that acquires a lock. Native methods must use the JNI interface to invoke other Java methods; thus, a program can be inspected for compliance with R3 by checking native methods that use the JNI interface.

3.3. Non-deterministic Read Sets

Shared memory among threads creates the possibility of deterministic commands reading different read-set values 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. We could keep track of all shared data or perform data race detection as in Eraser [6]. Generally the bookkeeping necessary to determine which objects are actually shared can result in a significant source of overhead: for example, an order of magnitude in time for Eraser.

We explore two restrictions to make this problem manageable. One is to assume R4A in Table 1, which requires every access to a shared variable be protected by a monitor (i.e., that the program is free of data races). Another way to achieve the same result is to assume R4B, which requires a run-time environment that enforces exclusive access to shared variables while a thread is scheduled (e.g., on a uniprocessor). Relaxing both restrictions for the general

case might require a combination of the approaches above and agreement on the shared data values.

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 through annotations or using statistical measures.

Our solution 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. Combined with identical initial values, identical lock acquisition sequences guarantee all commands executed by corresponding BEEs have identical read-set values.

Unfortunately, many real programs do not satisfy R4A: even the JRE provided by Sun does not meet this restriction for all shared data. In particular, static data members are often shared between threads without explicit shared method invocations. As BEE replicas reach agreement on the sequence of lock acquisitions, these data races can cause the state of the primary and backup to diverge, even when the races do not affect the semantics of the program.

Figure 1 shows a use of static data members without acquiring a lock. Object `shared_data`, a static data member, is shared by all `Example` objects. The guard on line 4 is not protected by a monitor, which allows different thread schedules at the primary and the backup to invoke `synchronized_method` a different number of times, preventing agreement on the sequence of lock acquisitions. Testing our implementation of replicated lock acquisitions required removing these race conditions in the JRE by hand! Although the code in Figure 1 contains a data race, we wanted to find a less labor-intensive way to handle this common (mal)practice.

We also consider an approach for handling shared data that does not rely on R4A, but assumes R4B instead. It eliminates non-deterministic read sets by replicating at the backup the order in which threads are scheduled at the primary. When R4B holds on a uniprocessor, the BEE whose thread is being scheduled effectively changes *all* its shared variables to local variables because no other BEE is allowed to execute commands. By replicating the order in which threads are scheduled, our implementation ensures that when R4B holds the order of access to shared data is replicated regardless of whether data races exist.

3.4. Output to the Environment

The state machine approach strives to hide replication from the environment by requiring 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.

Hiding replication of output is easy if the output is either *idempotent* or *testable*. In the former, 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. For example, seeking to an absolute offset in a file is an idempotent operation, while seeking to a relative offset is not. If the current offset can be read, a relative seek becomes a testable operation. Except for these cases, it is impossible to maintain the "single correct machine" abstraction in the presence of failures. 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. This impossibility result forces us to introduce a further restriction R5 in Table 1 that requires all native method output to the environment be either *idempotent* or *testable*.

Replication of volatile output might be necessary for correct operation. For example, the OS underneath the JVM is considered part of the environment. 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 could be restored simply by replaying the output (i.e., if the methods are idempotent), but volatile state generally requires special treatment. For instance, replaying messages on a socket would not recover the state at the backup because sending messages is in general not an idempotent operation. An extra layer must be added to make sending messages either an idempotent or testable operation.

Our protocol uses a novel interface, called *side effect handlers*, to replicate the lost volatile state of the primary. Native methods can 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 that replicate the volatile state of the primary created by the additional native methods. For example, through the interface we have included methods to handle file I/O in the standard JRE libraries. Restriction R6 in Table 1 requires applications to use this interface whenever they invoke a native method that creates volatile state.

4. Implementation

Sun's JVM provides two implementations of multithreading. The *native threads* version provides thread scheduling in the underlying OS, while the *green threads* version implements a user-level thread library for a uniprocessor inside the JVM. Since R4A depends upon the application's use of locks and not the low-level thread implementation, both libraries can take advantage of techniques that achieve replica coordination by replicating the sequence of

lock acquisitions. Indeed, multi-processor applications running with native threads on an SMP can take immediate advantage of the technique described in Section 4.2.

Enforcing R4B, however, requires changes in the thread library. Since our first goal is to maximize portability, we have focussed on implementing a replicated thread scheduler for green threads. Our approach could be extended to native threads (see [7])—we leave this as future work.

We add two system threads to the JVM. One performs failure detection to allow the backup to initiate recovery. The other is concerned with the transfer of logging information, either by sending it (at the primary) or by receiving it (at the backup). These additional threads join the several system threads that perform tasks such as garbage collection and finalizing objects. We next 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, composed of their class name, method name, and argument types, in a hash table. Generally, 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 raised are sent to the backup, which keeps an identical hash table. Before executing a method during recovery, the backup checks if it is stored in the hash table. If so, the backup always uses the corresponding return values and exceptions, whether or not it actually invokes the method. If the method is indeed invoked in order to reproduce volatile output, the backup discards the generated return values and exceptions. The side effect handlers discussed later provide an extra layer to handle specific cases where the return value may reflect volatile environment state (e.g. returning a file descriptor from a file open command).

4.2. Nondeterministic Read Sets

Data races and scheduling differences among the JVM's threads can make read sets containing 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 R4A 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* recording 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* recording the number of times lock l_id has been acquired so far.

These records are created by the primary, but they are used during recovery by the backup. Therefore, for each thread and lock, the primary needs to generate virtual t_ids and l_ids 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 might 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 a counter); 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, then 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, then t 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 acquires 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.

This approach only replicates the lock acquisition sequence, which may require extra synchronization when ordering is important. If multiple threads are interacting with the environment (e.g., reading or writing a log) and the interleaved order is important, then synchronization is required to ensure an identical order between the primary and the backup even if the synchronization is not required for correctness at the primary.

Replicated Thread Scheduling. The second approach relies on the assumption R4B that the scheduling lock protects all shared data. Whenever the primary interrupts the execution of a thread t to schedule a new thread, it sends a *thread scheduling record* to the backup, which uses it during recovery to enforce the primary's schedule. A record is comprised of (br_cnt, pc_off, mon_cnt, l_asn, t_id), where:

br_cnt counts the control flow changes (e.g., branches, jumps, and method invocations) executed.

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

mon_cnt counts the monitor acquisitions and releases performed by t.

l_asn records the lock acquisition sequence number when t is rescheduled while waiting on a lock.

t_id is the thread id of the next scheduled thread.

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. 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 can contend for the same locks.

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

1. 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.
2. 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 might not be rescheduled at the backup. It is easy to use *mon_cnt* to enforce the correct scheduling.

Threads can 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' might have interacted

[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.

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 can synchronously collect garbage by invoking a JRE native method. Asynchronous garbage collection is performed periodically by a garbage collector thread and during memory allocation when memory pressure indicates collection is needed. Since garbage is unused memory by definition, we initially avoided replicating the behavior of the asynchronous collector thread. However, asynchronous garbage collection can be a source of non-deterministic read sets. Indeed, both *soft references* and *finalizer methods* create paths for non-deterministic input to application threads.

Soft references are used to implement caches. By fudging the definition of garbage, the referenced objects 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 replicas, collection of soft references might occur at different times at different replicas. For instance, the primary might find an object in its cache, while the backup might 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, we use a much simpler solution: all soft references are simply treated as strong references, which represent active objects and are therefore never collected. This shortcut has no effect on our experiments because there is never enough memory pressure to dictate the collection of soft references.

Another possible source of non-determinism is improper use of finalizer methods. These methods are intended to allow objects to reclaim resources 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, allowing different behaviors at the primary and the backup. Our current implementation assumes that finalizer methods only free unused memory or perform other deterministic actions on local memory. 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 can 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 deal with output commands in native method through a novel approach based on what we call *side effect handlers* (SE handlers). 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 registers with the JVM information about the native methods that the handler will manage, including the signature of the method, whether the method is a non-deterministic command and/or an output command, and whether its arguments should be logged (i.e., if they are also output arguments).

test The backup calls this method 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. *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 The primary calls this method 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 might store the file descriptor and the amount written (or the current file pointer offset).

receive The backup calls this method to receive the state stored by the primary through the *log* method. Before saving the state, *receive* can compress it: for example, *receive* could compress the results of several file writes into one offset for the file pointer.

restore The backup calls this method during recovery. It is invoked only once. *restore* recovers the volatile state affected by output commands. If *receive* has compressed the results of multiple commands, *restore* might be able to recover the appropriate state directly instead of replaying the commands. For example, to recover an open file *restore* would open the file and set the file pointer to the appropriate offset.

Each SE handler can manage a set of related native methods. For example, we have one handler for all native file I/O methods. The handlers we have written for the standard libraries are automatically added to the system during startup. Applications can incorporate their own handlers using the same functions. Using SE handlers allowed us to add support for file I/O in the standard libraries. The same approach can be used by application writers to incorporate user-supplied output commands.

5. Experiments

Our experimental setup consists of two Sun E5000 servers, each with 15 400MHz UltraSPARC II cpus and

[2] Similar arguments also apply to *weak references* [8], which we treat similarly.

Table 2: Properties of benchmarks pertinent to our implementation.

Implementation	Event	jess	jack	compress	db	mpegaudio	mtrt
Both	Intercepted NM	64088	631295	419	96011	10031	1473
	NM Output Commits	763	34	102	703	10	133
Replicated Lock Acquisition	Logged Messages	4873592	12833046	2355	53492759	14717	701738
	Locks Acquired	4809503	12201750	1935	53396747	4685	700264
	Objects Locked	4515	505223	102	15612	21	161
	Largest l_asn	1410798	746136	633	5286641	1955	34738
Replicated Thread Scheduling	Logged Messages	64089	631296	420	96012	10032	30638
	Avg. Reschedules	0	0	0	0	0	29163

2GB memory running SunOS 5.8 connected by a 100 Mbps Ethernet. The primary runs on one machine and logs events at the backup running on the other. On performing an output, the primary waits until the backup acknowledges having logged all events up to the output event. The backup keeps its log in volatile memory.

Sun's JVM does not include the source code for Just-In-Time (JIT) bytecode compilation, which dynamically converts methods from bytecodes into native machine code instructions. Without the source we cannot use JIT compilation because we cannot include our modifications to some bytecode executions (e.g., interception of native method invocations). Hence, all of our experiments are performed in *interpreted mode* (i.e., without JIT compilation). JIT compilation reduces the execution time of CPU intensive code but has little effect on communication, which is our primary source of overhead. Hence, although the overhead on a JVM using JIT compilation is hard to predict, we believe that probably it wouldn't change significantly except for compress, which is 2 times faster on Sun's HotSpot JVM with JIT compilation. The other benchmarks vary from 20% faster to 20% slower execution time, probably resulting in comparable changes to the overhead.

To estimate the costs of adding fault tolerance to the JVM, we run the SPEC JVM98 benchmark on the replicated lock acquisition implementation, the replicated thread scheduling implementation, and the original Sun JVM. The programs in the benchmark vary widely in their characteristics. Compress is a CPU-intensive Lempel-Ziv compression application. Jack is a parser generator which is run on input to generate a parser for itself. Db contains a memory-resident database that is queried multiple times. Jess is an expert shell system that computes on a set of common puzzles with progressively larger rule sets. Mpegaudio decompresses MPEG-Layer 3 audio files. Mtrt is the only multi-threaded application in the benchmark and consists of a ray-tracer rendering a scene of a dinosaur. Though Mtrt is the only multi-threaded application, several other apps (notably Db) contain much synchronized code. We did not include results for javac (included in SPEC JVM98) because we could not get the application to run on Sun's original JVM.

Table 2 summarizes the properties of the benchmark applications with respect to our implementation. Database queries in Db result in the most lock acquisitions by far, while Jack locks more unique objects. All applications have few intercepted native methods and even fewer output commits. The largest l_asn shows that the lock acquisitions are skewed—few locks are responsible for most acquisitions. The average number of reschedules in the last row shows that though many locks are acquired in all of the benchmarks, only Mtrt actually requires them for multi-threading.

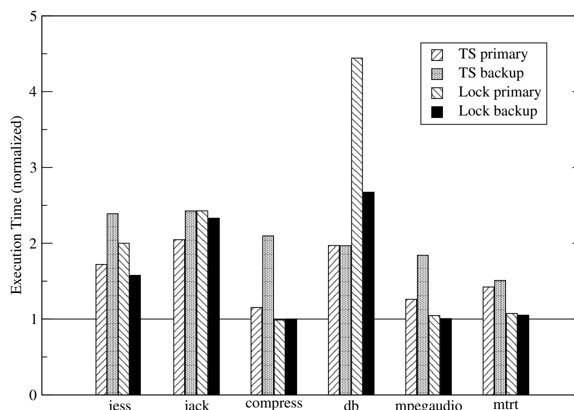

Figure 2: Comparison of our implementations using green threads normalized to our JVM without replication. The TS columns represent our replicated thread scheduler implementation, and the Lock columns represent the replicated lock acquisition implementation. The execution times of each benchmark are (in secs): jess (167), jack (182), compress(541), db (354), mpegaudio (419), mtrt (163).

We implemented replicated lock acquisition for both the green threads library supporting user-level threads on uniprocessors and the native threads library supporting multi-threading on an SMP. We only implemented thread scheduling for green threads. We found the overheads exhibited by the two implementations of replicated lock acquisitions to be qualitatively similar. We thus only report results from our implementation using green threads. All experiments are performed on lightly loaded machines running in multi-user mode; experiments were repeated until 95% confidence intervals were within 1% of the mean.

Figure 2 shows the overall execution times of the benchmark applications using each of our replication approaches normalized to the corresponding times without any replication. The primary columns are the execution times of the primary logging events to the backup, while the backup columns give the times for the backup to replay events from the log. Although our implementation was not tuned aggressively (we only optimized some in the replicated thread

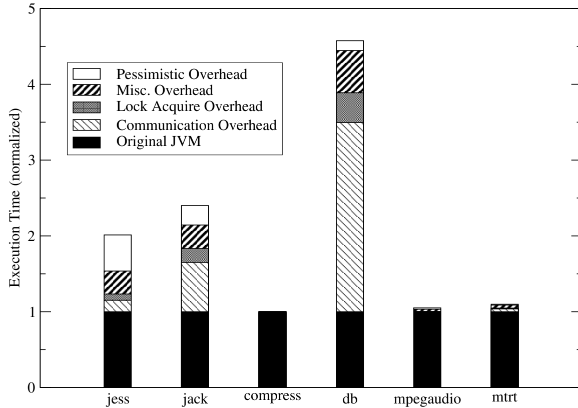

Figure 3: Normalized overhead for replicated lock acquisition implementation using green threads library.

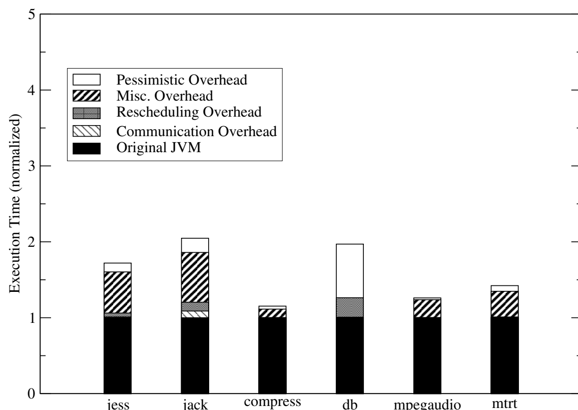

Figure 4: Normalized overhead for replicated thread scheduling implementation using green threads library.

scheduler), we observed under 100% overhead for most applications. Replicating lock acquisitions has an average of 140% overhead (skewed by Db) for green threads, well above the replicated thread scheduling's 60% average.

The overhead for replicated lock acquisitions (Figure 3) ranges from 5% (Mpegaudio) to 375% (Db). The large overhead in Db is a result of processing its more than 53 million lock acquisitions. In Figure 3, Communication Overhead represents the time spent sending messages to the backup, and Lock Acquire Overhead measures the time spent storing information on lock acquire. Pessimistic Overhead represents the time spent waiting for acknowledgments from the backup on output commit events.

In our implementation lock acquisition messages are very small (36 bytes). The primary buffers such messages and sends them to the backup either periodically or on an output commit; in the latter case, the primary sends the buffered messages and waits for an acknowledgment. Similarly, the backup only sends an acknowledgment message after processing a burst of incoming logging messages.

The sources of overhead for the replicated thread scheduling implementation are detailed in Figure 4. Communication Overhead and Pessimistic Overhead are as in Figure 3, while Rescheduling Overhead measures time spent updating counters and storing scheduling decisions. The overhead varies from 100% (Jack) to 15% (Compress).

Replicating thread scheduling yields a lower communication overhead than replicating lock acquisition: only Mtrt logs any thread schedule records to the backup. Further, to reduce the number of records created, a record is sent only when a new thread is scheduled. All other benchmarks are single-threaded; hence, they do not involve transmission of any records. The replicated lock acquisition implementation does not take advantage of this single-threaded case, sending many unnecessary messages.

For such applications, we expect replicated thread scheduling to incur smaller overhead than replicated lock acquisition. In practice, however, we observe that this is not always the case (see Figure 2), because storing thread progress incurs significant overhead. As seen in Figure 4, the overhead of replicated thread scheduling is dominated by the Misc. Overhead, which captures the overhead resulting from extra bookkeeping. In an earlier version of our implementation, the bookkeeping overhead for the replicated thread scheduler overwhelmed any communication advantages. To reduce these costs, we were forced to add about 12 instructions that update counters and keep track of the virtual machine's PC to the hand-written optimized assembly loop that executes bytecodes at the heart of the JVM. We believe significant additional reductions could be achieved by optimizing the code further. Also, using a deterministic scheduler as in the Jikes RVM [9, 10] or Jalapeño [11] might result in lower overhead substantially because the progress indicators would be simplified.

The two approaches to handling multi-threading present different tradeoffs. Replicating lock acquisitions may be less effective if a thread acquires or releases objects several times before being rescheduled. Further, replicating thread scheduling handles automatically the single-threaded case as no extra messages are sent. Nonetheless, replicating lock acquisitions is still a compelling approach because it works on multiprocessor systems, and may provide better performance, as in the case of Mtrt.

As communication overhead is the dominant source of overhead in our experiments, the amount of communication for a given application created by each technique is an effective predictor of their performance.

6. Related Work

Replica coordination[1, 2] can be implemented at any level of a system's architecture, from the application level [12]all the way down to the hardware [13]. Systems that implement replica coordination at intermediate levels include TFT [14] (at the interface above the operating system) and [3], in which replica coordination is implemented above a *virtual machine* that exports the same instruction set architecture as HP's PA-RISC.

We first reported on our fault-tolerant JVM in [15]. Since then, we have become aware of other concurrent and independent effort that address some of the same issues discussed in this paper. Basile and others report on replicating multi-threaded applications in [16]. They develop a leader-follower replicated lock acquisition algorithm that assumes

R4A and a Byzantine failure model for a webserver application. Their algorithm for replicated lock acquisition is similar to ours; however, they do not explore scenarios where R4A doesn't hold.

Recently, Friedman and Kama have also explored the idea of modifying the JVM (in their case, the Jikes RVM [9]) to achieve transparent fault-tolerance [10] using semi-active replication. Although we share the same goals, our approaches differ in three fundamental ways. First, their approach only applies to systems where R4A holds, while we explore multiple ways to handle the non-determinism introduced by multi-threading. Second, they do not address applications with non-deterministic native methods, though they do address I/O within the JRE. Finally, they report experiments using JIT, while all our experiments are performed in interpreted mode because we require access to the source code for JIT.

Earlier work on debugging multi-threaded applications addressed non-determinism. LeBlanc and Mellor-Crummey first introduced recording lock synchronization and shared memory accesses for debugging replay [17]. More recently, Choi and Srinivasan apply this approach to Java in the DejaVu tool for debugging assuming R4A in [18] and R4B in [11]. DejaVu records *logical thread intervals* wherein a thread performs non-deterministic events such as monitor entry/exit and shared variable accesses. The intervals include thread schedules for the underlying deterministic thread scheduler of the Jalpeño JVM.

As our focus is fault-tolerance, our implementation differs in several ways. First, we include a general approach to handling application-provided native methods. Second, DejaVu does not address output to the environment. Third, the Jalapeño scheduler reschedules at deterministic *yield points*, simplifying thread execution progress tracking.

Their trace sizes are much smaller than ours by clever use of intervals, but the overhead incurred is still 40%-80%, comparable to ours without pessimism. Our implementation could benefit from the use of intervals. For the multi-threaded Mtrt application there would only be 56 intervals instead of 700258 lock acquisitions—four orders of magnitude fewer events, resulting in a significant saving in space and probably also time.

To the best of our knowledge, replicating lock acquisitions for handling multi-threading was first proposed by Goldberg, et al., for Mach applications in [19]. When replicating lock acquisitions, correctness depends on the absence of data races. By augmenting the type system, Boyapati and Rinard developed race-free Java programs which meet R4A [20]. Data race detection mechanisms [21, 6] could also be used to verify R4A holds for a given program.

Our implementation of replicated thread scheduling is based on Slye and Elnozahy [7]. They record thread progress during normal execution using a count of control flow changes (branches, jumps, function calls). Our solution differs in two ways: 1) the JVM cannot track all control flow changes (e.g., while executing a native method) and 2) we do not recover all threads (e.g., the garbage collector).

7. Conclusions

We build a fault-tolerant JVM using the state machine approach. 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. Our results suggest that this is a viable solution for providing transparent fault-tolerance to Java applications.

References

[1] L. Lamport, "Time, clocks, and the ordering of events in distributed systems," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[2] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec 1990.

[3] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," in *Proceedings of SOSP 15*, Dec 1995.

[4] T. Lindholm and F. Yellin, *The Java™ Virtual Machine Specification, 2nd Ed.* Addison-Wesley, April 1999.

[5] S. Liang, *The Java™ Native Interface: Programmer's Guide and Specification.* Addison-Wesley, June 1999.

[6] S. Savage *et al.*, "Eraser: A dynamic race detector for multi-threaded programs," *ACM TOCS*, vol. 15, no. 4, pp. 391–411, October 1997.

[7] J. H. Slye and E. Elnozahy, "Support for sotware interrupts in log-based rollback recovery," *IEEE TOCS*, vol. 47, no. 10, pp. 1113–1123, October 1998.

[8] P. Chan, R. Lee, and D. Kramer, *The Java Class Libraries: 2nd Ed, Vol 1 Supplement for the Java™ 2 Platform, Std Ed, v1.2.* Addison-Wesley, June 1999.

[9] IBM, "Jikes RVM," 2002. [Online]. Available: http://www.ibm.com/developerworks/oss/jikesrvm/

[10] R. Friedman and A. Kama, "Transparent fault-tolerant JVM," Department of Computer Science, The Technion, Tech. Rep. CS-2002-19, Dec 2002.

[11] J.-D. Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides, "A perturbation-free replay platform for cross-optimized multithreaded application," in *Proceedings of IPDPS*, 2001.

[12] K. P. Birman, "The process group approach to reliable distributed computing," *Communications of the ACM*, vol. 36, no. 12, pp. 37–53, 1993.

[13] J. Bartlett, J. Gray, and B. Horst, "Fault tolerance in tandem computer systems," in *The Evolution of Fault-Tolerant Systems*, A. Avizienis, H. Kopetz, and J.-C. Laprie, Eds. Vienna, Austria: Springer-Verlag, 1987, pp. 55–76.

[14] T. C. Bressoud, "TFT: A Software System for Application-Transparent Fault Tolerance," in *Proceedings of FTCS 28*, June 1998, pp. 128–137.

[15] J. Napper, L. Alvisi, and H. Vin, "A fault-tolerant java virtual machine," University of Texas, Dept. of Computer Sciences, Tech. Rep. TR02-56, May 2002.

[16] C. Basile, Z. Kalbarczyk, K. Whisnant, and R. Iyer, "Active replication of multithreaded applications, Tech. Rep. UILU-ENG-02-2201, March 2002.

[17] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Transactions on Computers*, vol. C-36, no. 4, pp. 471–482, April 1987.

[18] J. Choi and H. Srinivasa, "Deterministic replay of java multi-threaded applications," in *SIGMETRICS Symposium on Parallel and Distributed Tools*, August 1998, pp. 48–59.

[19] A. Goldberg, A. Gopal, K. Li, R. Strom, and D. F. Bacon, "Transparent Recovery of Mach Applications," in *Usenix Mach Workshop*, 1990, pp. 169–183.

[20] C. Boyapati and M. Rinard, "A parameterized type system for race-free Java programs," in *Proceedings of OOPSLA*, Tampa Bay, FL, October 2001.

[21] G.-I. Cheng *et al.*, "Detecting data races in cilk programs that use locks," in *Proceedings of ACM SPAA*, 1998.