Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 113 (2005) 85–104
www.elsevier.com/locate/entcs

JVM Independent Replay in Java

Viktor Schuppan[1], Marcel Baur[2], Armin Biere[1]

Computer Systems Institute, ETH Zentrum RZ H, CH-8092 Zürich, Switzerland

Abstract

Deterministic replay can help to understand the cause of a failing execution of a multi-threaded program. Stepwise browsing of a counterexample serves the same purpose in the context of static and dynamic checking. In this paper we present a tool for deterministic replay of a multi-threaded execution of a Java program. The replay engine is independent of a specific JVM. We also suggest a language to describe thread schedules. Such schedules can be produced either directly by a tool or virtual machine or can, given some additional information, be extracted from a bytecode trace. Thus, off-the-shelf debuggers can be used for both, cyclic debugging of multi-threaded Java programs, and for browsing of concurrent execution traces produced by many checking tools. Experimental results show that correct replay can be performed with acceptable overhead across a number of virtual machines. Plug-ins have been implemented to generate schedules automatically for Java PathFinder and for JNuke.

Keywords: execution replay, debugging aids, verification tools, testing tools, Java

1 Introduction

Cyclic debugging [21] is still an important means to locate the source of a fault in a program. Non-determinism of any kind makes cyclic debugging difficult. In programming languages such as Java or C#, concurrency adds an important source of non-determinism. However, most debuggers provide limited support for debugging multi-threaded applications.

If sufficient information is collected during the execution of a multi-threaded application to ensure that the concurrent behavior can be deterministically reproduced, cyclic debugging can be used in the same manner as for sequential

[1] Email: {schuppan,biere}@inf.ethz.ch
[2] Email: baur@adbw.ch

doi:10.1016/j.entcs.2004.01.032

programs. Several solutions that *capture* and deterministically *replay* the execution of a multi-threaded application have been proposed for Java [5,10,15].

The above approaches provide an integrated solution for capture and replay. In this paper we present a separate replay facility. It can enforce a thread schedule as it might occur during the execution of a multi-threaded Java application on a uniprocessor. The replay facility is independent of a specific virtual machine, i.e., any compliant [20] virtual machine can be used for replay. Replay is achieved by instrumenting the class files of the application according to a given schedule and linking a replay engine during runtime.

A number of static and dynamic checkers produce error traces for concurrent Java programs, e.g., Java PathFinder [32] or Bandera [7]. Our replay facility can free developers of such tools to provide trace browsers or debuggers of their own but instead only produce information required for replay. Application developers can then use standard debuggers for trace browsing, thereby utilizing most features found in a debugger and working in the development environment they are used to.

An execution is described as a sequence of thread switches. The point of a thread switch is given as a combination of the address of an instruction and an execution count for that instruction. If that format is not produced directly by a virtual machine or verification tool, the required information can be extracted from a sequence of bytecode if thread identity and some information on the thread state are available.

We used a custom virtual machine to generate schedules. In addition, we implemented a listener to automatically extract schedules from an execution of error paths with Java PathFinder. Replay has been tested with several virtual machines. Single-stepping through an execution trace without the user having to select the active thread has been performed with `jdb`, Eclipse and others. The slowdown incurred by the replay facility is usually less than 10 times for Sun's virtual machine (v. 1.4).

Section 2 gives an overview on related work. The format of a thread schedule is detailed in 3. Section 4 explains the mechanisms used by our implementation. Results on the performance of the replay mechanism are stated in section 5. In section 6 we outline a more general format for thread schedules. The last two sections discuss open aspects and conclude.

2 Related Work

Most early approaches to replay address parallel executions. These require a different approach as reconstructing a schedule cannot ensure correct replay of parallel programs. Pan and Linton [24] record and replay the values of shared

objects at each access. LeBlanc and Mellor-Crummey implemented a scheme that records the order of accesses based on version counters in Instant Replay [19]. Tai et. al. [31] use a source-to-source transformation of an Ada program to replay a sequence of synchronization events. Their scheme assumes that shared variables are protected appropriately. Netzer [23] presents an algorithm that reduces the amount of data generated by these approaches.

Russinovich and Cogswell [26] reconstruct the scheduling of an execution on a uniprocessor by putting all but one threads to sleep to achieve replay for the Mach operating system. Our basic approach for replay and representation of a thread schedule is probably closest to theirs. They use a software instruction counter [22], i.e., the number of backward control transfers and the current instruction pointer to decide when a thread switch should happen, while we directly count the executions of an instruction. The implementation is based on a modified debug version of the Mach system libraries. An application needs to be compiled with a flag that keeps one of the processor registers free to hold a counter for the backward control transfers. Then, as in our approach, assembly files are instrumented, linked with a debug library, supplied with a schedule file, and can be executed in an existing debugger.

A first solution to recording deterministic replay of executions of multi-threaded Java programs was proposed by Choi and Srinivasan [5]. They introduced the notion of a logical thread schedule, given by a linear order of the critical events during an execution. Their implementation is based on a modified Java virtual machine. The approach was extended to include record/replay of networking events in distributed applications [17]. Later [4], DejaVu has been integrated into the Jalapeno virtual machine with a focus on ensuring symmetric behavior for record and replay of the entire virtual machine in the presence of cross-optimization of the application and the runtime environment. Their work is different from ours in that they try to reconstruct precisely the behavior of a single virtual machine while we want to achieve portable replay across different virtual machines. Our approach lacks precision in comparison with DejaVu. This seems acceptable as the traces we want to replay are produced by dynamic or static checking tools that typically represent failures that are not based on internal behavior of a virtual machine. However, this view might change if tools are available that also consider potential problems caused by the Java memory model [20].

In [10] a capture and replay algorithm was implemented as part of a tool to support testing of concurrent Java programs. Besides debugging, replay is used to automatically test whether different interleavings at a specific execution point lead to different results if a data race is present. Replay is performed with bytecode instrumentation, the scheduling information is given by thread

switches. A number of improvements seems to have been made in comparison with DejaVu [5], however, the paper gives few details on the mechanisms used in their implementation.

JaReC [15] also aims at portable replay for multi-threaded Java applications. JaReC uses bytecode instrumentation to capture and replay schedules based on Lamport clocks for objects. It assumes a data-race free program and, therefore, only needs to instrument at synchronization operations. This technique has originally been used by the same group to implement RecPlay [25], a tool that combines record/replay and data race detection for Solaris. JaReC is similar to our approach in that it instruments an application and performs calls to a replay engine at runtime. However, instrumentation is performed on the fly by the virtual machine. This enables dynamic loading and replaying of classes over a network and prevents having several versions of a class. If present, JaReC uses the JVMPI [30] for instrumentation. While concurrent schedules can be represented, it is less flexible for sequential thread schedules as thread switches can only be specified at the start or end of synchronized sections. It is not clear from [25] whether recursive locks can be handled by their mechanism.

Choi and Zeller [6] showed how a combination of replay and delta debugging [33] can be used to locate the source of an error by finding two schedules with a small set of differences, one leading to a failure and the other producing a correct result.

We only cover replay of non-determinism introduced by concurrency. A description of a prototype of a tool for Java that replays other forms of nondeterminism is given by Steven et. al. [28]. In principle, our schedule format is flexible enough such that additional commands to replay input/output behavior or random numbers could be incorporated.

3 Representing Information for Replay

In this section we describe our approach to represent the information required for replay. After discussing different possible solutions we briefly outline our approach. An example illustrates our schedule format before details are given.

3.1 What Information is Needed for Replay?

Parallel or concurrent execution typically introduces nondeterminism into a program by allowing accesses to shared objects happen in different order. If this is the only form of nondeterminism, replay can be achieved by restoring Lamport's "happens-before" relation [18] of an execution, i.e., the partial order of events that causally affect each other.

Some approaches record the order per relevant event using vector clocks [23] or version counters for shared objects [19]. Logging can be restricted to the order of synchronization operations if the program is assumed to be free of data races [31,25,15]. On a uniprocessor a more implicit approach can be taken: it is (largely) sufficient to restore the scheduling as it happened during the original execution [26]. For Java, extra information on the interaction of `interrupt` and `notify` or time-out of `wait`, `sleep`, or `join` may need to be provided. In between is the approach by Choi and Srinivasan [5]. They record/replay a logical thread schedule, i.e., a linear order of all relevant events.

An approach that directly records the order of relevant events seems more elegant in that it reflects the true reason for a failure or a different result. It is also, to some extent, more abstract and might, thus, be better suited for further analysis. It retains parallelism inherent in the application even if executed originally on a uniprocessor. On the other hand, restoring a thread schedule will often need to log less data. In addition, capturing a thread schedule is probably easier to implement as might be extracting a thread schedule from a bytecode trace. Our primary concerns were simple extraction of schedules from checking tools and stepwise execution of a trace in a debugger. We therefore have implemented replay based on thread schedules. A thread schedule is represented as a sequence of thread switches. The point of a thread switch is given as a location in the program and an execution count of the bytecode at that location.

3.2 Example

Figures 1 – 3 give a producer-consumer scheme as an example. The `main` method starts one producer thread and two consumer threads; then it waits for those to terminate. Neither clients nor buffer use synchronization. Fig. 1a) shows a schedule that leads to a failure. The main thread (id 0) starts its children and hands over to the producer thread (id 1) that, in turn, processes up to the point where the buffer contains one element; it stops before the bytecode in line 13 is executed. Next, the first consumer (id 2) thread queries whether the buffer contains an element, gets a positive answer but switches to the second consumer (id 3) before it gets to actually perform the `get` in lines 9/12. The second consumer can complete a full iteration and stops only before executing the `goto` in line 13. The failure occurs, when the first consumer continues. Figure 1b) shows use of loops. The producer and one of the consumers interleave for two iterations. Then, producer and the second consumer interleave; this continues infinitely.

```
# 0 (main) running
before Prodcons 1 36 1
switch 1 # p
before Producer 1 13 1
switch 2 # c1
before Consumer 1 9 1
switch 3 # c2
before Consumer 1 13 1
switch 2 # c1
# error when executing get
```

a) Schedule leading to a failure.

```
# 0 (main) running
before Prodcons 1 36 1
switch 1 # p
loopbegin
loopbegin
before Producer 1 13 1
switch 2 # c1
before Consumer 1 13 1
switch 1 # p
loopend 2
```

```
loopbegin
before Producer 1 13 1
switch 3 # c2
before Consumer 1 13 1
switch 1 # p
loopend 2
loopend inf
```

b) Infinitely looping schedule.

Fig. 1. Schedules for Producer-Consumer

```
public class Producer extends Thread {
    public void run() {
        while (true) {
            if (Buffer.notFull()) {Buffer.put(0);}
    }   }
} // Producer
```

```
public class Consumer extends Thread {
    public void run() {
        int i;
        while (true) {
            if (Buffer.notEmpty())
                {i = Buffer.get();}
    }   }
} // Consumer
```

```
public class Prodcons {
    public static void main(String argv[]) {
        Producer p;
        Consumer c1, c2;
        p = new Producer();
        c1 = new Consumer(); c2 = new Consumer();
        p.start();
        c1.start(); c2.start();
        try {
            p.join();
            c1.join(); c2.join(); }
        catch (InterruptedException ie) {}
        System.exit(0); }
} // Prodcons
```

```
public class Buffer {
    private static int buffer, count;
    static boolean notFull() {return count < 1;}
    static boolean notEmpty() {return count > 0;}
    static void put(int i) {
        buffer = i;
        count++;
        }
    static int get() {
        if (!notEmpty()) {
            System.err.println("Error.");
            System.exit(1); }
        count--;
        return buffer; }
} // Buffer
```

Fig. 2. Source code for Producer-Consumer

```
Method void run() // Producer.java
   0 goto 3
   3 invokestatic #2 <Method boolean notFull()>
   6 ifeq 3
   9 iconst_0
  10 invokestatic #3 <Method void put(int)>
  13 goto 3
```

```
Method void run() // Consumer.java
   0 goto 3
   3 invokestatic #2 <Method boolean notEmpty()>
   6 ifeq 3
   9 invokestatic #3 <Method int get()>
  12 istore_1
  13 goto 3
```

Fig. 3. Bytecode for Producer-Consumer

3.3 A Language for Schedules

A text-based format is used. The schedule contains one command per line. Blank lines are ignored. Comments start with a #-sign and extend to the end of the line. The following types of commands are supported [3]:

events: `before` and `in` indicate when an action should occur,

[3] Note, that the term *event* is used in our language description with a slightly different meaning than in the rest of the paper: here, an event is an indication of the point in time at which a steering action should be taken, while in the other parts an event is something that is to be replayed or controlled.

actions: `switch`, `notify`, `timeout`, `die`, `terminate`, and `log` trigger actions of the replay engine, and

control flow: `loopbegin` and `loopend` specify loops in the schedule.

An event command can be followed by several action commands. One thread is executing at a time. Initially, the thread corresponding to the `main` method of the application is running.

An action might occur before or after a particular bytecode is executed. In addition, for blocking invocations of `wait`, `join`, and `sleep`, a thread switch can take place after the active thread has entered a blocking state. Keywords `before` and `in` are used to specify that an action occurs before execution of a bytecode or after it has blocked, respectively. An action after the execution of an instruction is specified as happening before executing the next bytecode. However, the latter does not exist if a bytecode terminates the current thread either by returning from a `run` or the `main` method of the application or by throwing an exception not caught by that thread. But both cases cannot lead to a blocking state; therefore, the `before` command is used with `die` or `terminate` to specify these situations. A bytecode is identified by the class name of the method containing it, that method's index in the class, and the offset within that method. The last position states the count of the attempted executions of that particular bytecode immediately before or during which the action is performed (in other words, one more than the number of completed executions). The execution count is reset after each event.

A thread switch suspends the active thread and normally resumes another. The target is given by a `switch`, a thread is identified by a unique numerical id assigned successively in the order of creation. Id `none` is used as target if some threads are alive but all of them blocked – then, no thread is unblocked.

A Java thread may be in one of several states. Apart from executing or being ready to execute, it can be waiting for the lock of an object, joining another thread or sleeping. A waiting thread may already be notified. In all blocking states it may be interrupted or timed out. Sun's Java documentation [16,20,29] does not specify unambiguously the transitions between different states. For details on the thread model we use, see Fig. A.1 in the appendix. The sequence of thread switches does not give all information needed for precise replay of a concurrent execution. A thread reacts differently to a call to `interrupt` depending on whether it is still blocked (it throws an exception when it restarts execution) or has already been notified or timed out (it sets a flag). Actions `notify` and `timeout` can be used to track the state accordingly. These actions need only be stated in the schedule if an interaction between `interrupt` and `notify` or time-out occurs – otherwise replay will be also correct without giving this information explicitly.

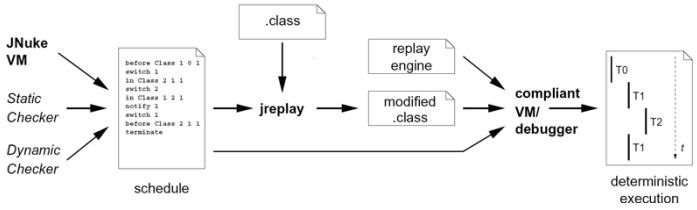

Fig. 4. Overview of jreplay

If a thread terminates, a thread switch has to be performed. However, the dying thread must not be blocked. A command `die` is used in this case. The replay engine waits for the dying thread to terminate until it hands control to the next thread. A `terminate` command can be used to unblock all threads and disable replaying. If performed before the proper end of execution, this may leave the application in an inconsistent state. If the end of a schedule is reached without executing `terminate`, the last switched to thread continues. Finally, a `log` command can be used to print messages.

The format supports nested loops in the schedule. For $i \geq 1$, the body of a loop between `loopbegin` and `loopend i` is executed i times. `loopend inf` generates an infinite loop that may be used to represent, e. g., counterexamples obtained from a model checker. A brief summary of the syntax is given in appendix B.

4 Mechanisms

Replay of a schedule is achieved by instrumenting the original application according to a given schedule and supplying a library containing a replay engine in addition to the instrumented class files to the virtual machine at start-up. No separate thread is created to control replay. Rather, each application thread performs method calls to the replay engine at appropriate times. Given a proper schedule, every compliant [20] Java virtual machine should then reproduce the original result. See Fig. 4 for an overview.

To replay a schedule the replay engine ensures that only the thread specified by the schedule may proceed in its execution. The remaining threads are kept in a blocked state. Each location in the application where an action should occur at some point in the schedule is instrumented with a call to the replay engine. The engine checks with the schedule whether the next location and the specified counter value has been reached. If this is the case, actions following the event command are performed. In the case of a thread switch the next thread according to the schedule is unblocked by the current thread that thereafter blocks itself.

```
public void block() throws Exception {
    synchronized(lock) {
        while (blocked) {
            try {lock.wait();}
            catch (InterruptedException e)
                {/* report error */} }
        blocked = true; }
} // block
```

```
public void unblock() {
    synchronized(lock) {
        blocked = false;
        lock.notifyAll(); }
} // unblock
```

Fig. 5. Blocking and unblocking a thread

The routines `block` and `unblock` are shown in Fig. 5. A thread blocks by waiting on an object `lock`, it is unblocked by a call to `notifyAll` on that object by a different thread. If the thread switch is not caused by a call to `wait` in the application, the lock used is an object privately allocated to that particular thread by the replay engine. Therefore, only the desired thread will wake up. A boolean flag `blocked` prevents that a thread actually performs the `wait` in `block` only after another thread has already issued the corresponding `unblock`.

When a thread calls `wait` on an application object, it releases all (potentially recursive) locks on that object. A thread can wait on one object at a time. To retain that semantics of a thread switch induced by a `wait` in the application the wrapper for `wait` in the replay engine therefore temporarily replaces the private `lock` object of a thread with the object the application thread is waiting on; the original call to wait in the application is discarded (but arguments are checked for exceptions), the actual call to `wait` is performed by `block` as shown in Fig. 5. [4] Calls to `notify` and `notifyAll` in the application are replaced with calls to wrappers in the replay engine that check only for exceptions. The actual notification happens as part of a thread switch in `unblock` only when a thread switches back to one of the threads waiting on an object: `unblock` notifies the object a thread is waiting on. All threads waiting on that object will receive a notification, but `blocked` will allow only the target of the thread switch to continue. A blocked thread starts processing again when it is the target of a thread switch, i.e., time-outs that can be specified in a call to `wait` are ignored.

Calls of the application to `join` and `sleep` are also substituted with invocations of the replay engine. Exceptions are checked for and a thread switch is performed if indicated by the schedule. Again, timing is not replayed.

The effect of a call to `interrupt` differs between a situation when the interrupted thread is blocked due to a `wait`, `sleep`, or `join` or when it is caught outside of these routines. Calls to `interrupt`, `interrupted`, and

[4] An invocation of `wait` may throw an `IllegalArgumentException`, an `IllegalMonitorStateException`, or an `InterruptedException`. We check for exceptions in that order, according to the behavior of Sun's virtual machine (JDK 1.4.1_02) for Linux.

`isInterrupted` are therefore replaced with calls to the replay engine. At the level of the virtual machine neither notification nor joined state are the same during replay as during the original execution. Instead, the state of a thread is tracked in the replay engine, the substitutes to `interrupt`, `interrupted`, and `isInterrupted` use that information to replay the original behavior. Not emulated is behavior w.r.t. interruptible channels and Selectors [29].

A new thread is registered with the replay engine and assigned a numerical id when an instance of `Thread` or a subclass is created. At the beginning of a `run` method of a class that subclasses `Thread` or implements `Runnable` a call to the replay engine is inserted that blocks the starting thread when it is called by this thread for the first time.

When a `terminate` command is processed in the schedule all threads are unblocked. Thus, threads that were blocked due to an invocation of `wait` or `join` will continue with their execution even if they should remain blocked according to the original trace. If the schedule ends without a `terminate`, the replay engine remains active, as does the last thread that was switched to; threads that were left blocked remain blocked.

Our implementation provides no special support for thread switches within classes that are part of the Java API or used in its implementation (i.e., `java.*`, `javax.*`, `sun.*`). More specifically, the replay engine is not reentrant and the instrumentation facility does not take special care of methods that are substituted by the replay engine. In addition, the implementation of the Java API differs between vendors. Ensuring portable replay would therefore require a translation of the locations of thread switches when using different implementations for schedule generation and replay. We do not regard this as a severe restriction as many checking tools treat classes that are part of the Java API as "safe" and do not perform thread switches within these classes, the obvious exception being calls by the application to `wait`, `sleep`, `join`, and `yield` that are handled separately by our implementation.

The Java memory model [20] allows an implementation not to resynchronize data between the private memory of a thread and shared memory for some operations. Only when locks are taken or released a read or write operation to shared memory is required. As the replay engine uses locks, it might alter the behavior of the original application in these cases. Replay may also not be precise due to different implementations between virtual machines or different optimizations performed by a virtual machine on the original and instrumented code [5].

[5] Jikes RVM produced different results in the LUFact benchmark for original and instrumented versions. A similar effect could be achieved by inserting calls to methods (not referring to application logic) in the original code.

5 Experiments

In this section we briefly describe how to capture execution traces on the fly. We then report on a series of experiments that indicate performance and portability of our replay engine. Finally, we mention an extension of Java PathFinder that generates schedules for replay.

5.1 Capturing Thread Schedules during Executions

A schedule can be generated easily on the fly during an execution of a Java application if the virtual machine offers a means to invoke a user-defined routine after each executed bytecode. The virtual machine needs to provide information on the last and next threads and the last and next bytecodes of the last thread as well as on the state of each thread. It then suffices to count the number of times each bytecode is executed between two events and to log corresponding events and actions. For proper replay thread switches, calls to `notify`, time-outs, thread deaths and deadlocks must be included. A corresponding procedure is given in Appendix C.

5.2 Capturing and Replaying Benchmarks

5.2.1 The JNuke Framework

The replay engine and instrumentation facility are part of the JNuke project of the formal methods group at ETH. Goal of JNuke is to develop a framework for dynamic and static analyses of Java bytecode. As first component a virtual machine for Java was implemented [13] that allows rollback of execution steps to explore different thread schedules as was done by Rivet [3]. On top of the VM, the Eraser algorithm [27] is used to check for data races. An algorithm to dectect high-level data races [1] has already been implemented. Work in progress aims at performing rollback also for input/output events.

Capturing of thread schedules can be done simply by employing the feature of this VM to implement a custom thread scheduler. We used a simple round robin scheduler in JNuke that performs a thread switch after a fixed number of instructions. This allows to select frequency and restrict locations of thread switches. The former proved useful to generate thread switches at arbitrary locations during testing of the replay engine. The latter is still used to prevent thread switches in Java library classes.

Capturing with the custom VM still requires considerable overhead compared to unconstrained execution on Sun's JVM. This was the most limiting factor to capture thread schedules of longer executions.

5.2.2 Results

The replay engine was tested with Jikes RVM v. 2.2.1, Kaffe Virtual Machine v. 1.0.7, SableVM v. 1.0.8, Sun's virtual machines shipped with Java 2 SE v. 1.3.1-06, and with Java 2 SE v. 1.4.1-02. Platform was a PC with a single Intel PIII-800 running Linux 2.4.19.

The benchmarks are taken from the multi-threaded part of the Java Grande Forum Benchmark Suite [12]. Section 1 measures performance of low level threading mechanisms, i.e., `wait/notify`, accessing `synchronized` objects and methods, and thread creation, start, and destruction. Section 2 contains compute-intensive programs with varying levels of synchronization: while LU-Fact uses barriers frequently, Crypt needs no synchronization other than start and termination of worker threads.

Tables 1 and 2 give the results, where n indicates the number of threads. For each virtual machine, the run time of the uninstrumented program (in seconds), the run time needed for replay of a given thread schedule on the same VM, and the overhead for the replay are given. We do not have a facility to capture a schedule on an arbitrary virtual machine. Therefore, the schedule for replay has been generated with JNuke and is the same for all virtual machines. However, the schedules that lead to the run time reported for the uninstrumented application might not be the same as that replayed. Run times are averaged over three runs. Note, that the runtime for some of the applications with smaller execution time may vary up to a factor of 2 between two executions.

Except for Sun's 1.4 JVM the virtual machines produced errors or timed out with Fork/Join from section 1. We therefore do not report results here. Due to limitations of our virtual machine at the time of writing we could not generate a trace for the Series benchmark in section 2.

The overhead for replay of a specific schedule depends on the frequency of calls to the replay engine and on the number of thread switches performed. Note, that the number of thread switches might differ between original and replayed executions. The SOR benchmark uses a barrier with busy waiting for synchronization. The replayed execution probably spends less time there. In most cases the slowdown of the replayed execution is less than a factor of 10. This seems acceptable for interactive debugging and is in the same range as the figures reported by [15].

The simple scheduler we used may insert thread switches at arbitrary locations in the application. Thus, for some of the examples a call to the replay engine is performed before every bytecode of the original instructions in core routines. The sizes of the class files containing these core routines grow correspondingly – 7 bytecode instructions are inserted per call to the replay engine.

Benchmark	n	Sun JVM 1.4			Sun JVM 1.3			Jikes RVM			Kaffe			SableVM		
		orig.	replay	ovrh.	orig.	replay	ovrh.	orig.	replay	ovrh.	orig.	replay	ovrh.	orig.	replay	ovrh.
Barrier	2	6.1	124.3	20.5	err	136.2	err	9.5	99.4	10.4	2.2	410.5	179.4	21.4	787.7	36.6
	4	22.9	157.2	6.8	err	172.7	err	9.4	121.5	12.7	2.7	482.7	174.1	69.2	942.7	13.6
	8	34.5	191.0	5.4	err	203.0	err	9.1	143.8	15.7	2.8	501.0	174.3	90.9	1000.9	10.9
Sync	2	15.1	13.1	.8	17.0	20.7	1.4	8.9	30.4	3.3	1.7	19.3	10.8	5.6	144.6	25.3
	4	14.8	10.9	.7	19.8	17.4	.8	8.7	28.7	3.2	1.5	17.2	11.3	err	123.8	err
	8	18.8	9.9	.5	15.9	15.9	1.1	8.6	27.2	3.1	1.3	14.5	10.3	4.4	107.4	24.2

Table 1
Results for section 1 of Java Grande Forum Multi-Threaded Benchmarks

Benchmark	n	Sun JVM 1.4			Sun JVM 1.3			Jikes RVM			Kaffe			SableVM		
		orig.	replay	ovrh.	orig.	replay	ovrh.	orig.	replay	ovrh.	orig.	replay	ovrh.	orig.	replay	ovrh.
LUFact	2	3.1	34.8	11.0	63.2	57.2	.9	11.3	67.0	5.8	5.1	73.8	14.3	15.8	562.1	35.3
	4	7.3	33.6	4.4	126.0	56.1	.4	11.4	64.8	5.6	4.8	74.9	15.3	24.6	529.5	21.4
	8	15.7	33.5	2.0	503.7	55.1	.1	11.4	61.7	5.3	4.9	67.9	13.6	46.6	495.2	10.5
Crypt	2	5.6	20.2	3.5	5.8	25.7	4.4	13.7	40.0	2.9	5.6	23.9	4.2	18.4	135.2	7.2
	4	5.6	19.9	3.5	5.8	25.5	4.3	13.7	39.9	2.8	5.6	23.6	4.2	18.4	135.1	7.2
	8	5.6	19.8	3.5	5.8	26.5	4.5	13.7	40.2	2.9	5.6	24.3	4.3	18.5	136.0	7.2
SOR	2	4.3	3.5	.7	2.5	4.4	1.7	10.2	23.3	2.2	2.3	4.7	2.0	err	err	err
	4	8.8	3.8	.4	4.6	4.4	.9	12.3	22.5	1.8	4.3	4.4	1.0	err	err	err
	8	18.7	6.9	.3	9.3	10.4	1.1	17.0	26.4	1.5	8.6	11.3	1.3	err	err	err
SparseMatmult	2	4.0	24.9	6.0	4.0	37.5	9.0	12.9	53.9	4.1	5.2	43.4	8.2	16.3	339.8	20.7
	4	3.5	24.3	6.8	3.4	36.9	10.4	12.3	51.7	4.1	4.8	44.1	9.0	15.7	338.4	21.3
	8	3.1	23.8	7.4	3.0	35.9	11.5	12.0	54.6	4.5	4.5	42.0	9.1	15.6	341.0	21.7

Table 2
Results for section 2 of Java Grande Forum Multi-Threaded Benchmarks

However, most class files are not affected by replay and, thus, do not change.

The instrumentation of an application preserves the line number table attributes of the original version. This enables transparent source level debugging. A given thread schedule can be replayed using, for example, stepwise execution. Apart from setting breakpoints no special interaction is required. The debugger automatically selects the only thread ready to run for execution. Tests have been successful using Sun's jdb, Eclipse [9], JDebugTool [8], and JSwat [14].

5.3 A Listener for Java PathFinder

Java PathFinder [32] is an explicit-state model checker for Java developed at NASA Ames. Since release 3, Java PathFinder provides listener interfaces that enable external tools to track and control the execution of an application and the search for a counterexample.

In a further experiment, we implemented a listener for Java PathFinder v. 3.1.1 to generate schedules during an execution. The implementation follows the algorithm in Fig. C.1. The listener handles `wait`, `notify`, and `notifyAll`. Time-out and sleep are not explicitly modeled by PathFinder, their implementation of `join` is based on `wait`. Among others we used a modified FundManager example from the PathFinder web page [2].

```
schedulebegin 0 # main
before Prodcons 1 36 1
send 1 0
receive 0 -1
scheduleend
```

```
schedulebegin 1 # p
receive 0 0
before Producer 1 13 1
send 2 0
receive 1 -1
scheduleend
```

```
schedulebegin 2 # c1
receive 1 0
before Consumer 1 9 1
send 3 0
receive 3 0
before Consumer 1 13 1
send 2 0
receive 2 0
scheduleend
```

```
schedulebegin 3 # c2
receive 2 0
before Consumer 1 13 1
send 2 0
receive 3 -1
scheduleend
```

Fig. 6. Schedule leading to failure in message-based format

6 A More Flexible Solution

The switch-based format presented in Sect. 3 is limited to a sequence of thread switches as performed on a uniprocessor. Most of the mechanisms described in Sect. 4 can be reused to implement a more flexible solution that retains parallelism if possible. That format still does not reflect directly the data dependencies of an execution. However, it seems flexible enough to handle the thread dependencies generated by many capturing algorithms.

A given "happens-before" relation [18] can be enforced by (non-blocking) sending and (blocking) receiving of messages between threads. Each thread has its own sequence of commands. The `switch` command is replaced with `send` and `receive` primitives. `send` sends to a thread with the given id a `long` value as message. `receive` waits for a message from a specific thread with a specific content. Messages are buffered at the receiver. Figure 6 shows the schedule leading to a failure from Fig. 1a) in the message-based format. A replay engine for the message-based format has been implemented as a prototype.

The format described in Sect. 3 can be converted to the message-based format by splitting a single schedule into a sequence of commands for each thread and replacing each `switch` command with a pair of `send` and `receive` operations. A conversion from the message-based format to a more specific instance in the switch-based format can be performed by a topological sort of the dependency graph given by the happens-before relation. Edwards [11] proved that minimizing the number of thread switches during the transformation of a concurrent control flow graph of an Esterel program is NP-complete. The same construction can be applied to a message-based thread schedule, for a proof see the full version of this paper available from the authors.

7 Discussion

In principle it would have been possible to use an `after` instead of a `before` primitive. Instrumentation is more difficult in that case: a call to the replay engine needs to be inserted before every potential successor of a bytecode after which a thread change occurred. On the other hand, it should not make much

difference for a trace generating tool to produce the required information: if a bytecode has been executed its successor will be known. However, if a schedule is extracted from a (partial) bytecode trace additional hints might be needed to find the proper successor for the last bytecode stated for a given thread.

In our schedule format the exact point of an event is uniquely determined by the location of a bytecode and the number of its (attempted) executions. If a schedule is generated during execution of the program this typically requires counting the number of executions of each bytecode since the last event. Alternatively, a software instruction counter [22] could have been used. During capture this would require only a single counter that counts the number of backward branches in application code. In our format, each instruction needs to be counted. On the other hand, during replay a software instruction counter requires to instrument each backward branch, while our format needs instrumentation only at points where an actual call to the replay engine should occur. Given the exponential complexity of some verification algorithms, a software instruction counter might we worth exploring.

The format of a schedule is somewhat verbose. While easily readable, a more compact representation might be necessary to represent long traces.

Source code debugging is preferable in most cases. Transparency can be achieved in this case (see Sect. 5.2.2). However, some concurrency errors will only be understandable at bytecode level. Thus, transparent stepping through bytecode would be desirable as well. This could be implemented by providing an additional attribute in class files that tells a debugger which bytecodes belong to the original application and which have been added or removed for replay.

8 Conclusion

We present mechanisms that allow deterministic execution of a thread schedule on an arbitrary JVM by instrumenting class files of an application and linking with a replay engine. A language to describe thread schedules is proposed. An execution on a uniprocessor is described as a sequence of thread switches. We show how to generate thread schedules during an execution of a Java program with the JNuke and Java PathFinder VMs. The overhead for replay execution is less than 10 times in most cases, which is good enough for interactive debugging. Several off-the-shelf debuggers are used for replay. Future work includes extending the format and replay engine to further events such as random numbers or input/output. We plan to make the replay engine available at `http://www.inf.ethz.ch/personal/schuppan/jreplay`.

Acknowledgement

We thank Cyrille Artho for helpful discussions, in particular, reminding us of recursive locks. Thanks also go to Pascal Eugster as the author of the virtual machine at the core of JNuke and for providing hooks to capture schedules.

References

[1] Artho, C., K. Havelund and A. Biere, *High-level data races*, Journal on Software Testing, Verification & Reliability (STVR) **13** (2003).

[2] Automated Software Engineering Group at NASA Ames Research Center, *Fund managers example*, http://ase.arc.nasa.gov/visser/jpf/jdc.html.

[3] Bruening, D., "Systematic Testing of Multithreaded Java Programs," Master's thesis, MIT (1999).

[4] Choi, J.-D., B. Alpern, T. Ngo, M. Sridharan and J. Vlissides, *A perturbation-free replay platform for cross-optimized multithreaded applications*, in: *IPDPS*, 2001.

[5] Choi, J.-D. and H. Srinivasan, *Deterministic replay of Java multithreaded applications*, in: *SPDT*, 1998.

[6] Choi, J.-D. and A. Zeller, *Isolating failure-inducing thread schedules*, in: *ISSTA*, 2002.

[7] Corbett, J., M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby and H. Zheng, *Bandera: Extracting finite-state models from Java source code*, in: *ICSE*, 2000.

[8] debugtools.com, *debugtools.com – JDebugTool graphical Java debugger*, http://www.debugtools.com/.

[9] eclipse.org, *Eclipse*, http://www.eclipse.org/.

[10] Edelstein, O., E. Farchi, E. Goldin, Y. Nir, G. Ratsaby and S. Ur, *Framework for testing multi-threaded Java programs*, Concurrency and Computation: Practice and Experience **15** (2003).

[11] Edwards, S., *An Esterel compiler for large control-dominated systems*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **21** (2002).

[12] EPCC, *The Java Grande Forum benchmark suite*, http://www.epcc.ed.ac.uk/javagrande/index_1.html.

[13] Eugster, P., "Java Virtual Machine with Rollback Procedure allowing Systematic and Exhaustive Testing of Multi-threaded Java Programs," Master's thesis, ETH Zrich (2003).

[14] Fiedler, N., *JSwat graphical Java debugger*, http://www.bluemarsh.com/java/jswat/index.html.

[15] Georges, A., M. Christiaens, M. Ronsse and K. D. Bosschere, *JaReC: a record/replay system for multi-threaded Java applications*, in: *Parallel Computing: Grids and Applications*, 2002.

[16] Gosling, J., B. Joy, G. Steele and G. Bracha, "The Java Language Specification," Addison-Wesley, 2000, second edition.

[17] Konuru, R., H. Srinivasan and J.-D. Choi, *Deterministic replay of distributed Java applications*, in: *IPDPS*, 2000.

[18] Lamport, L., *Time, clocks, and the ordering of events in a distributed system*, Communications of the ACM **21** (1978).

[19] LeBlanc, T. J. and J. M. Mellor-Crummey, *Debugging parallel programs with Instant Replay*, IEEE Transactions on Computers **C-36** (1987).

[20] Londholm, T. and F. Jellin, "The Java Virtual Machine Specification," Addison-Wesley, 1999, second edition.

[21] McDowell, C. and D. Helmbold, *Debugging concurrent programs*, ACM Computing Surveys **21** (1989).

[22] Mellor-Crummey, J. M. and T. J. LeBlanc, *A software instruction counter*, in: *ASPLOS*, 1989.

[23] Netzer, R. H. B., *Optimal tracing and replay for debugging shared-memory parallel programs*, in: *Workshop on Parallel and Distributed Debugging*, 1993.

[24] Pan, D. and M. Linton, *Supporting reverse execution of parallel programs*, in: *Workshop on Parallel and Distributed Debugging*, 1988.

[25] Ronsse, M. and K. D. Bosschere, *RecPlay: A fully integrated practical record/replay system*, ACM TCS **17** (1999).

[26] Russinovich, M. and B. Cogswell, *Replay for concurrent non-deterministic shared-memory applications*, in: *PLDI*, 1996.

[27] Savage, S., M. Burrows, G. Nelson, P. Sobalvarro and T. Anderson, *Eraser: A dynamic data race detector for multithreaded programs*, ACM TCS **15** (1997).

[28] Steven, J., P. Chandra, B. Fleck and A. Podgurski, *jRapture: A capture/replay tool for observation-based testing*, in: *ISSTA*, 2000.

[29] Sun, *Java 2 Platform, Standard Edition, v 1.4.1 API Specification*, http://java.sun.com/j2se/1.4.1/docs/api/index.html.

[30] Sun, *Java Virtual Machine Profiler Interface (JVMPI)*, http://java.sun.com/j2se/1.4.1/docs/guide/jvmpi/jvmpi.html.

[31] Tai, K.-C., R. H. Carver and E. E. Obaid, *Debugging concurrent Ada programs by deterministic execution*, IEEE TSE **17** (1991).

[32] Visser, W., K. Havelund, G. Brat and S. Park, *Model checking programs*, in: *ASE*, 2000.

[33] Zeller, A. and R. Hildebrandt, *Simplifying and isolating failure-inducing input*, IEEE TSE **28** (2002).

A Thread States

Figure A.1 shows the thread model of the replay engine. The model is based on Sun's Java documentation [16,20,29]. Ambiguities for the `wait`-related transitions were removed based on results of a number of experiments with Sun's virtual machine, version 1.4.1_02 for Linux. For some `join`- and `sleep`-related transitions we could neither validate nor invalidate our assumptions by using only test cases.

B Schedule Syntax

`before <class_name> <method_index> <bytecode_offset> <execution_count>`

Prevents advancing to the next command in the thread schedule until the

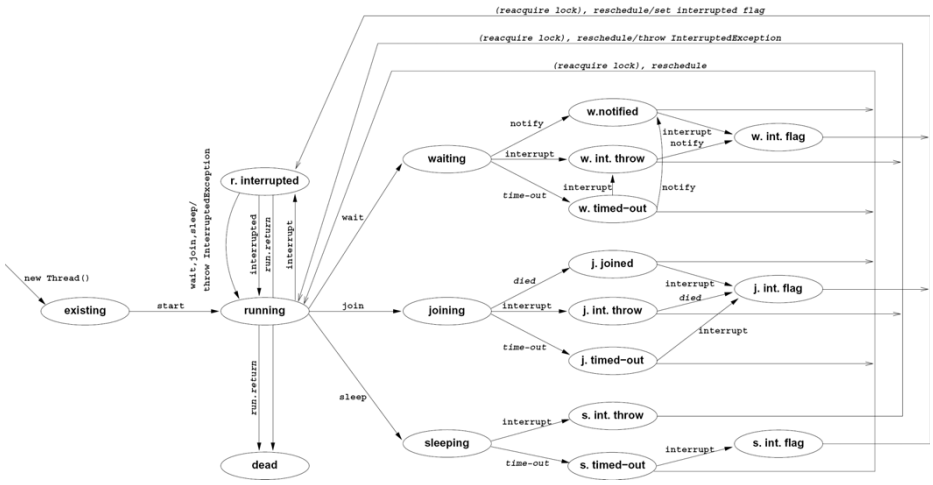

Fig. A.1. Model of thread states in Java used by the replay engine

specified bytecode was about to be executed for `execution_count` times since the last `before` or `in` command. Unsuccessful attempts to execute a bytecode count as one execution. As a consequence, a thread switch from a thread, that was about to execute a *monitorenter* instruction, but does not succeed, and immediately (without executing an instruction) switches to a different thread, is specified with an execution count of 1. On the other hand, if a thread switch from a thread occurs at the same location as the previous thread switch from that thread, performing a single execution of the specified bytecode between, an execution count of 2 is needed.

`in <class_name> <method_index> <bytecode_offset> <execution_count>`

Prevents advancing to next command in the thread schedule until the specified bytecode is blocked in the `execution_count`-th execution since the last `before` or `in` command. Otherwise same semantics as for `before`.

`switch <thread_id>`

Blocks current thread and, if `thread_id` is not `none`, unblocks thread `thread_id`.

`notify <thread_id>`

Processes a notification event in the state model of the replay engine. Does not directly affect state of the original application.

`timeout <thread_id>`

Processes a time-out event in the state model of the replay engine. Does not

directly affect state of the original application.

`die <thread_id>`

Lets the current thread terminate and, if `thread_id` is not `none`, unblocks thread `thread_id` after that.

`terminate`

Unblocks all threads and disables replay engine.

`log <log_level> <message>`

Prints `message` if `log_level` is less than or equal to the value of the Java system property `jreplay.verbosity`. Legal values for `log_level` are 1 – 4, `message` extends up to the end of the line or a # sign. Sequences of white space are represented as a single blank.

`loopbegin`

Marks the beginning of a loop. Encompasses all commands up to the next `loopend` of the same nesting depth.

`loopend [<i>|inf]`

Marks the end of a loop. The loop is executed `i` times if $0 < i \leq Long.MAX\_VALUE$, an infinite number of times if $i = inf$.

C Generating Schedules

Figure C.1 shows a procedure in a Java-like syntax to generate a schedule on the fly from an execution of a Java application. We use a similar algorithm in our experiments with Java PathFinder. If the algorithm is to be applied to bytecode traces, one will typically have to infer the bytecode that would have been executed after the last bytecode given for a thread, the state of that thread after the execution of that last bytecode, targets of calls to `notify` and time-outs.

```
HashMap count; /* assume that count returns 0 if a key is not yet
                  stored; use array semantics below */
boolean flag;  /* only print before command once */

printlnBefore(Instruction insn, int count) {
  if (!flag) {
    println(``before `` +
            insn.getFQClassName() + `` `` +
            insn.getMethodIndex() + `` `` +
            insn.getOffset() + `` `` +
            count);
    flag = true;
  }
}

printlnIn(Instruction insn, int count) {
  /* as ``before'', but ``in'' and no flag */
}

/* called after each execution of a bytecode. */
instructionExecuted(JVM jvm) {
  Thread t = jvm.getLastThread();
  Thread nextt;
  Instruction insn = t.getLastInstruction();
  Instruction nexti;

  flag = false;
  count[insn]++;
  if (insn instanceof NotifyInstruction) {
    printlnBefore(insn, count[insn]);
    println(``notify `` + ((NotifyInstruction) insn).getTarget());
  }
  if (jvm.hasNewTimedoutThread()) {
    printlnBefore(insn, count[insn]);
    foreach newly timedout thread tt do
      println(``timeout `` + tt.getId());
  }
  if (jvm.hasNextThread()) {
    nextt = jvm.getNextThread();
    if (t != nextt) {
      if (t.isAlive()) {
        if (!t.isWaiting() && !t.isJoining() && !t.isSleeping())
          printlnBefore(next, count[next] + 1);
        else
          printlnIn(insn, count[insn]);
        println(``switch `` + nextt.getId());
      } else {
        printlnBefore(insn, count[insn]);
        println(``die `` + nextt.getId());
      }
      count.clear();
    }
  } else if (jvm.isDeadlocked()) {
    if (t.isAlive()) {
      if (!t.isWaiting() && !t.isJoining() && !t.isSleeping())
        printlnBefore(insn, count[insn]);
      else
        printlnIn(insn, count[insn]);
      println(``switch none'');
    } else {
      printlnBefore(insn, count[insn]);
      println(``die none'');
    }
  } else {
    printlnBefore(insn, count[insn]);
    println(``terminate'');
  }
}
```

Fig. C.1. Procedure to generate a schedule from an execution