Portable record/replay for Java

A. Georges, M. Christiaens, M. Ronsse, K. De Bosschere
ELIS, Ghent University
St.Pietersnieuwstraat 41, B-9000 Gent
{ageorges, mchristi, ronsse, kdb}@elis.rug.ac.be

I. Introduction

A very common debugging technique is *cyclic* debugging. Here the programmer repeats the execution of the application and slowly zooms in on the part where he thinks an error may occur. Paramount to this scheme is that the erroneous execution can be replayed at will. Unfortunately, this is not always the case. In multi-threaded applications several threads may interact with each other in a non-deterministic manner. This may be because of e.g. synchronization operations between different threads.

It is our goal to eliminate the non-determinism caused by the synchronization operations performed by the application threads. To obtain this goal, we have implemented a record/replay system [2], [5] for Java, which we call JaRec. JaRec will trace the order of the synchronization operations during the record phase and enforce that same order during the replay phase.

Our main concern was the portability of the system. Hence, we have implemented the bulk of JaRec in Java, while making as little modifications to the JVM as possible. In fact, the only real modification required involves adding some code to the JVMPI library. Many existing replay system need to adapt the underlying JVM [1], [2], [6].

Furthermore, it is necessary to keep the intrusion during the record phase as small as possible, since that will ensure we record a representative execution.

II. Synchronization races

The non-determinism caused by synchronization operations is due to the fact that so-called synchronization races occur. If two (or more) threads are trying to obtain a lock on an object, we say they are racing to acquire that lock. The outcome of such a race is not predetermined. Hence, across several executions, it is not always the same thread that wins the race and obtains the lock first. Clearly, this can affect the outcome of the program, since manipulations of the same data structures may be done in a different order across different executions.

III. Synchronization in Java

In Java, there are several ways to perform a synchronization. First of all, there is the `synchronized` statement, that can be used in the body of a method. On the other hand, we have the `synchronized` attribute, that can be set on both a static and a non-static method. Essentially, the action taken for the synchronization is similar. In all three cases, an object is associated with the synchronization operation. The thread performing this synchronization obtains a lock on that object, and releases the lock when leaving the synchronized region. The scope of the synchronization is the entire method in the case of the `synchronized` attribute, whereas in the case of the `synchronized` statement, it is limited to the statement block.

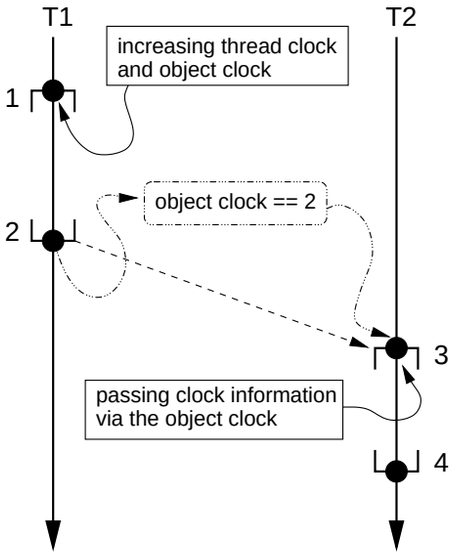

Fig. 1. The record phase.

If a thread $\mathtt{T}_i$ holds the lock of an object `O` guarding a synchronization operation, no other thread $\mathtt{T}_j$ may enter a synchronized region guarded by the same object `O`, since its lock is already held by $\mathtt{T}_i$. Hence, $\mathtt{T}_j$ must wait before the synchronization operation until the lock on `O` is released by $\mathtt{T}_i$.

IV. Record phase

We have opted for an ordering based record/replay system, where the order of critical event is traced. Because it is infeasible to record the order of all the instructions that are executed, due to (i) the extremely high intrusion this causes, (ii) the multi-threaded nature of applications and (iii) the weak memory-consistence model the JVM uses, we limit ourself to recording the order of the synchronization operations executed by the various application threads. During the record phase, JaRec keeps track of these synchronization operations. To impose order on them, JaRec uses Lamport clocks [4]. These are simply integers, repre-

senting logical clock values. Each time a thread T executes a synchronization operation with an associated object O, both the thread clock value and the object clock value are increased according to the following scheme:

$$\mathtt{LC}_{\mathtt{new}} = \max\left(\mathtt{LC}_{\mathtt{T}_i}, \mathtt{LC}_O\right) + 1 \quad \rightarrow \quad \mathtt{LC}_{\mathtt{T}_i} = \mathtt{LC}_{\mathtt{new}}$$

$$\text{and} \quad \mathtt{LC}_{\mathtt{O}} = \mathtt{LC}_{\mathtt{new}}.$$

Each thread keeps track of the clock values it obtains ($\mathtt{LC}_{\mathtt{T}}$) in a trace file. The clock values assigned to objects ($\mathtt{LC}_O$) are simply used to pass clock values from one thread to another. This is illustrated in Figure 1

V. Replay phase

During the replay phase, all threads fetch their clock values from the trace. When a synchronization operation is executed by a thread $\mathtt{T}_i$, its clock is set to the next clock value found in the trace for `T`. If a thread `T` tries to execute a synchronization operation, JaRec will first check if there are no other threads with a smaller clock value. If there are, then `T` must wait until it has the lowest clock among all threads. When more threads have the smallest clock value at a certain time, they may proceed concurrently.

VI. Instrumentation

As we do not want to change the JVM itself, because of portability issues, we will instrument the Java class files to run JaRec. Before the classes are loaded into the JVM, the are given by the JVMPI to the Profiler Agent. We have implemented an Agent that transfers these classes to another JVM, where they are instrumented end sent back. The instrumented classes are then passed on to the JVMPI by the Agent and loaded into the JVM executing the application. If the JVM does not offer a JVMPI, it is usually not too hard to add some communication code.

This scheme is entirely transparent to the JVM. Of course, caution must be exercised, since some classes may not be altered because they are hard coded into the JVM. We do not instrument such classes, e.g. `java.lang.Object`. Figure 2 illustrates the instrumentation and trace storage setup. The instrumentation itself involves wrapping each `monitorenter` and `monitorexit` instruction, so the record/replay system is called when synchronization operations are performed. Since these instructions do not appear in a method with the `synchronized` attribute, we replace these method with equivalent ones, where the body is contained in a `synchronized` statement block.

JaRec will also replace the `wait()` calls with wrapper code to emulate the effect of releasing the lock and competing for the re-acquisition after the `wait()`. We take care to keep the overhead as small as possible, as not to interfere with the execution of the application. In the replay phase, we add slightly different instrumentation code and we remove the `monitorenter` and `monitorexit` instruction, as well as the calls to a `wait()` method. In this phase, we arrange for the synchronization to be arranged entirely by JaRec.

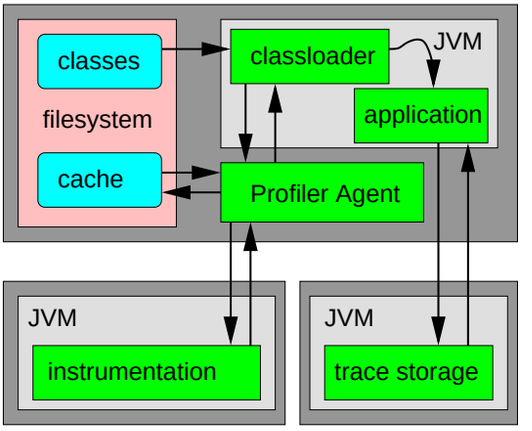

Fig. 2. The JVM setup for instrumentation and trace storage.

Instrumenting the bytecode sequences is done using the BCEL [3]. This library offers the possibility to change the Java classes on a high level of abstraction.

VII. Results

We have tested our implementation on a number of benchmarks. The overhead incurred for a synchronization operation is about a factor of three. Because most applications do useful things besides synchronizing, the average overhead during an execution is usually quite acceptable. The intrusion depends on the percentage of instrumented code that is executed and on the thread behaviour. When treads keep on synchronizing in an intertwined manner there is not much opportunity to store the clocks acquired in a space-efficient manner, causing a large overhead in the transmission of the clock values. During the replay phase, the overhead is higher, due to the constant halting and restarting of threads as they have to wait for the right clock value before they are allowed to proceed. Here too, there is overhead incurred by the transmission of the clock values.

References

[1] J.-D. Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *15th International Parallel and Distributed Processing Symposium (IPDPS'01)*, San Fransisco, CA, April 2001.

[2] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *ACM Sigmetrics Symposium on Parallel and Distributed Tools SPDT98*, pages 48–59. ACM, August 1998.

[3] M. Dahm. Byte code engineering. In *Java-Informations-Tage*, pages 267–277, 1999.

[4] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[5] M. Ronsse and K. De Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.

[6] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jrapture: A capture/replay tool for observation-based testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 158–167. ACM Special Interest Group on Software Engineering, ACM Press, NY, USA, 2000.