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

1 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 [1, 4] 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. 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.

2 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.

3 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 given to 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.

If a thread $\mathtt{T}_i$ holds the lock of an object `O` associated with a synchronization operation, no other thread $\mathtt{T}_j$ may enter a synchronized region with the same associated 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 $\mathtt{T}_i$.

4 Record phase

During the record phase, JaRec keeps track of the synchronization operations. To impose order on these operations, JaRec uses Lamport clocks [3]. These are simply integers, representing 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:

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

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.

5 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.

6 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, they are sent by the JVMPI to another JVM, where they are instrumented and sent back. The instrumented classes are then loaded into the JVM executing the application. 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.

The instrumentation itself involves adding byte code instructions to each `monitorenter` and `monitorexit` instruction, to the `invokevirtual` instructions calling a `wait` or `notify` method, wrapping synchronized methods, ... The instrumentation is done using the BCEL [2].

7 Measurements

We have tested our implementation on a number of benchmarks. The overhead incurred for a synchronization operation is about a factor of five. Of course, most applications do useful things besides synchronizing. Hence the average overhead during a complete run is normally quite acceptable. Since the overhead during the record phase is solely caused by the instrumentation, it actually depends on the percentage of executed instructions that are part of the instrumentation code. 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.

References

[1] 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.

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

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

[4] 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.