JaRec: Record/Replay for Multi-threaded Java Programs

Mark Christiaens Stijn Fonck Dries Naudts Michiel Ronsse
Koen De Bosschere
Department ELIS
Ghent University
Sint-Pietersnieuwstraat 41
Gent 9000, Belgium
{mchristi, sfonck, dnaudts, ronsse, kdb}@elis.rug.ac.be

1 Introduction

Debugging programs is hard but debugging multi-threaded programs is even harder. The main reason for this is that multi-threaded programs are usually non-repeatable due to races. A simple example of a race is shown in Figure 1. We see two executions of a program consisting of two threads, $T1$ and $T2$. Thread $T1$ adds 2 to the variable A while $T2$ multiplies A by 2. Depending on which thread executes faster, there are two possible results. Either $A = (3 + 2) \times 2 = 10$ or $A = (3 \times 2) + 2 = 8$. If we were to continue from here, a totally different execution could result from the different content of the variable A.

This non-repeatability makes life very hard during debugging. Programmers usually repeat the program execution and focus in detail on some parts that may be at fault. This technique is called *cyclic debugging*. The main precondition of cyclic debugging is that a faulty program will always reproduce the same error over and over again. When dealing with multi-threaded programs this property of repeatability is usually lost.

In order to solve this problem, a technique called record/replay was developed [2, 3, 4]. It consists of two phases: a record phase and a replay phase. During the record phase, a trace is generated that contains information on the order of the interactions between two threads. This trace can then be used during the replay phase to reproduce the exact same interaction between threads as occurred during the record phase. A major goal during the record phase is very low intrusion. During the replay phase, every debugging trick is allowed since intrusion can no longer alter the execution of the program.

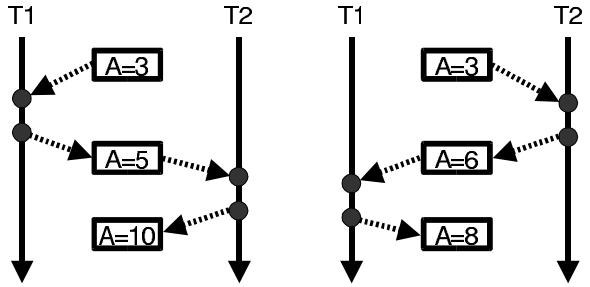

Figure 1: A simple race

2 JaRec

Our record/replay approach is based on a construct called 'Lamport clocks' [1]. In Figure 2 the essence is shown. We see three Java threads, $T1$, $T2$ and $T3$. These threads enter and exit two monitors using `monitor_enter`, `monitor_exit` or through calls to synchronized member functions. Entering and exiting a monitor is indicated by the opening and closing brackets. The order in which these monitors are entered is indicated by the arrows between the brackets.

We assume that all the accesses to shared objects are correctly synchronized using these monitors i.e. that there are no *data* races. There might still be races between threads trying to obtain a lock (*synchronisation* races). By forcing the order in which threads enter the monitors, we will have assurance that the whole execution will be replayed exactly as it was recorded. The order in which threads gain access to the monitors is stored in a trace file using Lamport clock values.

The Lamport clock values are also shown in Fig-

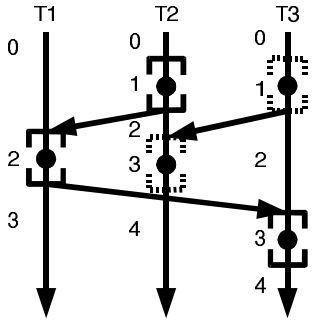

Figure 2: Lamport clocks obtained during the record phase

ure 2. Every thread is assigned a Lamport clock which is initialized to time 0. Each time a monitor is entered or exited, we verify which thread accessed the monitor previously. The Lamport clock of the current thread is updated to be 1 larger than the maximum of its own current Lamport value and that of the previous thread accessing the monitor. These consecutive Lamport clock values are stored in the trace file. During replay, when multiple threads are trying to enter a monitor, only the thread with the smallest Lamport clock value is allowed to proceed.

Both the record phase and the replay phase in JaRec are being implemented using the Java Virtual Machine Profiler Interface (JVMPI). We use the JVMPI to intercept a class when it is being loaded by the JVM. It is then transferred, over a socket, to a second JVM which replaces every `monitor_enter`, `monitor_exit` and synchronised member function by a call to our instrumentation routines. Then the class is returned to the original JVM and actually loaded.

We are building two types of instrumentation: one for the record phase and one for the replay phase. During record phase, our instrumentation is responsible for calculating and logging the Lamport clocks while during replay it is responsible for forcing the order in which threads gain access to the monitors. A second responsibility of the replay phase is to check whether the assumption that there are no data races is indeed correct. If this assumption does not hold, it will be flagged to the user who can then adjust his program.

Due to this data race detection the replay phase has usually been a magnitude slower than the original execution. This is mainly caused by the fact that every read operation and write operation is observed in isolation in order to detect data races. Using our approach, we will be able to instrument classes as a whole and to remove many of the unnecessary checks that were repeated time and time again for every instruction. A simple example occurs when a field in an object is accessed several times in a loop without intervening synchronisation. It is clear that if the first access to this field is a data race, then all following accesses will also be data races. Therefor, we can limit the amount of checks to just the first access in the loop.

3 Conclusion and Work in Progress

Multi-threaded programs are in general non-repeatable due to races. This makes debugging multi-threaded programs very hard. We are building a record/replay system which will allow programmers to replay an execution of a multi-threaded Java program. We achieve this by instrumenting a Java program on-the-fly using the JVMPI interface. We are currently finishing the record phase and starting to implement the replay phase of our system.

References

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

[2] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[3] Robert H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Summary of the ACM/ONR workshop on Parallel and distributed debugging*, pages 1–11. ACM, May 1993.

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