

Record/Replay in the Presence of Benign Data Races

Mark Christiaens
Elis, Ghent University
Sint-Pietersnieuwstraat 41
9000 Ghent, Belgium
email: mchristi@elis.rug.ac.be
tel: +32 (0) 9 264 33 67
fax: +32 (0) 9 264 35 94

Jong-Deok Choi
IBM T. J. Watson Research Center
P.O. Box 704, Yorktown Heights
N.Y. 10598, USA
email: jdchoi@watson.ibm.com

Michiel Ronsse
Elis, Ghent University
Sint-Pietersnieuwstraat 41
9000 Ghent, Belgium
email: ronsse@elis.rug.ac.be

Koen De Bosschere
Elis, Ghent University
Sint-Pietersnieuwstraat 41
9000 Ghent, Belgium
email: kdb@elis.rug.ac.be

Abstract *In this article we present our experience with the integration of record/replay in the Jalapeño virtual machine. The goal of record/replay is to be able to faithfully replay an application. Previous work in Jalapeño focused on the replay of Java applications on uni-processors. Here we describe additional work done to obtain replay with low intrusion on multi-processor systems by doing ‘ordering based’ record/replay. During ordering based record/replay we only record the order of the synchronization operations performed. A prerequisite of this technique is that there are no data races present in the application that is to be replayed. However, we found that Jalapeño contains many benign data races. A major contribution of this article is that we show how one can circumvent these data races and still perform a meaningful replay of the application.*

Keywords: record/replay, data races, debugging, parallel applications, Java

1 Introduction

Cyclic debugging is a commonly used technique to find bugs. It consists of repeatedly executing a buggy application and slowly zooming in on the bug. After each debug-iteration, more accurate assumptions about the bug can be made until eventually it is located and corrected.

The underlying assumption of cyclic debugging is that there is no non-determinism in the application i.e. that one is able to reproduce the faulty behavior at will. This assumption is often not met. Many applications for example consume input that cannot be reproduced easily e.g. mouse events, network activity, ... If we wish to reproduce bugs in such applications, we need to record its input [1].

In the last decade, a new programming technique has become prevalent which introduces another type of non-determinism: multi-threading. Multi-threaded applications can produce different results even when provided with the same input. This non-determinism is caused by ‘races’. In Figure 1 we see the two types of races we will consider.

In Figure 1 on the left, we see two threads, T_1 and T_2 , that are modifying a common variable, A . We see that there are two possible outcomes of this interaction: variable A can either be 1 or 2 depending on the timing between the two threads. The threads are said to be ‘racing’ on the data in the variable A hence the term ‘data race’. Data races are usually an oversight of the programmer and should be removed.

On the right, we see a similar situation but now the accesses to the variable A are protected by a lock/unlock

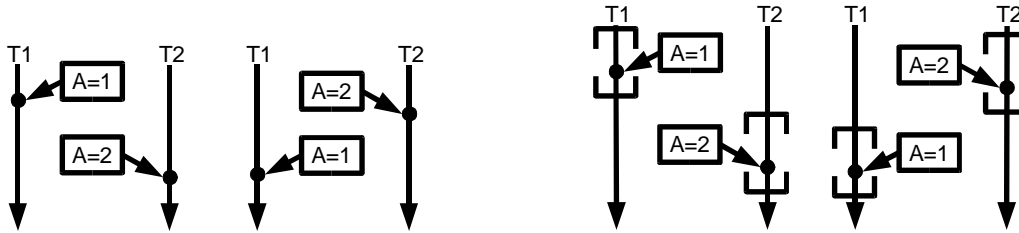


Figure 1: Example of the two classes of races: on the left we see a data race, on the right a synchronization race.

pair (indicated by the square brackets). Here, the threads are still racing but this time to gain access to the lock. Once we know which thread has obtained the lock first, all other operations will happen in a known order. This situation is called a ‘synchronization race’. A synchronization race is usually an intentional construct by the programmer.

Data races and synchronization races both introduce non-determinism in the execution of an application. A known technique to remove this non-determinism and therefore ease the debugging effort is called ‘record/replay’[2, 3]. Record/replay consists of at least two phases: a record phase and a replay phase.

During the record phase, all sources of non-determinism are recorded (input and thread interaction). The major goal during the record phase is low intrusion. During the replay phase the recorded information from the record phase is used to force all non-deterministic events to occur in the same way as during the original execution. The primary goal is to obtain a faithful replay of the original execution. This might involve a high level of intrusion. During replay, most debugging tricks that are too intrusive to use during a normal execution are allowed since they can no longer alter the execution of the application.

We have implemented the record/replay technique in Jalapeño [4], a research virtual machine capable of executing a large subset of the Java language. The record/replay approach we’ve taken is known as ‘ordering based record/replay’. It records the order of all the synchronization operations happening in the application and enforces these orderings again during replay. If there are no data races present in the application, this approach is sufficient to ensure a faithful replay.

Jalapeño already has a record/replay infrastructure called DejaVu [5]. It is capable of replaying the whole operation of the Jalapeño (the JIT-compiler, the memory subsystem, ...) but it was designed to do replay when Jalapeño is run on a uni-processor.

In what follows, we will show how we implemented record/replay for Jalapeño on a multi-processor. Furthermore, we found during our implementation, that Jalapeño contains many benign data races. These were often introduced to improve performance but are a showstopper when it comes to ordering based replay. We show that it is possible to still do ordering based record/replay even in the presence of data races.

In Section 2 we will explain the concept of ordering based replay. In Section 3 we show how this can be implemented in the presence of data races. In Section 4, we present an overview of the practical implementation in Jalapeño of the above mentioned concepts. In Section 5 we give some conclusions.

2 Ordering based record/replay

There are in essence two approaches to record/replay: ‘content based’ and ‘ordering based’ record/replay. Content based record/replay [6] tries to observe a thread in isolation. It will record all data read from main memory and feed this data back to the application during replay. Due to the enormous data bandwidth involved, this approach is seldom taken.

Instead, we chose to implement ordering based record/replay [2, 3, 7, 8]. Ordering based record/replay tries to exploit the fact that the data read by one thread is usually produced by that same thread or by other threads in the application. By re-executing all threads simultaneously and forcing the order of their interaction, we can reproduce most of the data needed by each thread in isolation. Still, due to the bandwidth involved, recording the order of all operations of all threads is not a practical option. What is more, many modern processors use a very weak memory model which no longer allows to establish the order between *all*

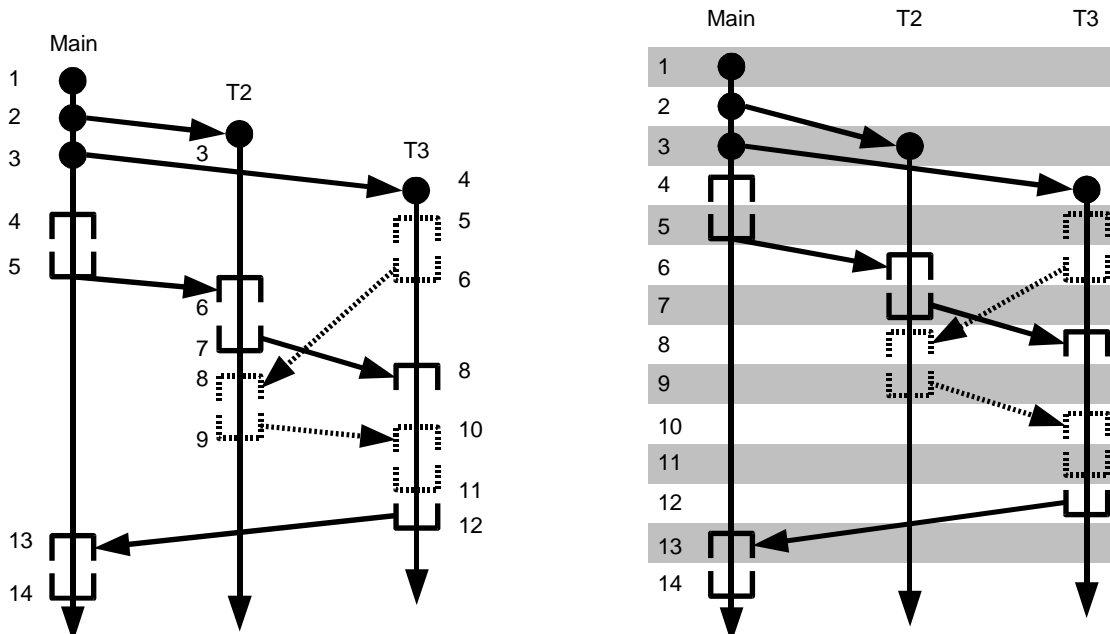


Figure 2: Lamport clocks: on the left during the record phase, on the right during the replay phase.

events executed on two different threads. However, using these memory models it is still possible to establish the order between synchronization operations. That's why we've chosen to only record the order between synchronization operations.

Frequently, ordering based record/replay is done using 'Lamport clocks' [9] as illustrated in Figure 2. Every thread and every synchronization object (a semaphore, a mutex, ...) is assigned a private Lamport clock (an integer). The first thread to be started will be assigned a Lamport clock with value 1. All synchronization objects will initially be assigned a Lamport clock of 0. Now, each time a thread, T , performs a synchronization using an object, O , the Lamport clocks of both, $LC(T)$ and $LC(O)$, are adjusted as follows:

$$LC_{new}(T) = LC_{new}(O) = \max(LC_{old}(T), LC_{old}(O)) + 1 \quad (1)$$

This process as it occurs during the record phase is illustrated on the left of Figure 2. The resulting Lamport clock values corresponding to the synchronization operations are stored into a trace file and are used during the replay phase to steer the re-execution as illustrated in Figure 2 on the right. During the replay phase, the trace file is used to order the synchronization operations. A thread is only allowed to proceed with a synchronization operation if the synchronization operation will occur with a Lamport clock value that is smaller or equal than any pending synchronization operations.

As can be seen in Figure 2, during replay Lamport clocks result in additional wait dependencies between the threads and therefore a loss of parallelism. For example, during the record phase, the `Main` thread and `T3` were allowed to execute their synchronization operation with resp. Lamport clock values 4 and 5 in parallel. During replay, these synchronization operations will be forced to happen one after the other. Still, in the past [2] it was shown that by using Lamport clocks the size of the trace file can be reduced substantially. This results in a low overhead during the record phase which was considered more important than a low overhead during replay.

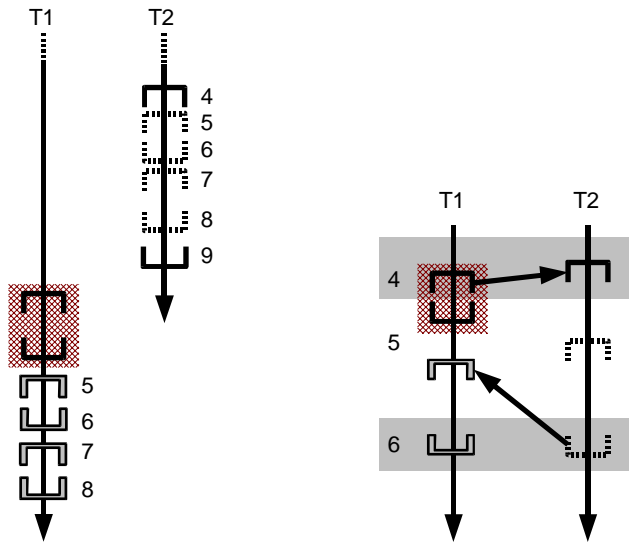


Figure 3: Example of a deadlock caused by the use of Lamport clocks in the presence of benign data races.

3 Record/replay in the presence of benign data races

3.1 Benign data races

The main precondition when doing ordering based record/replay using only the synchronization operations is that the order between the synchronization operations is sufficient to force all other operations to happen in the same order. This is clearly not always the case. Consider again Figure 1 on the left. There is no synchronization whatsoever present in this example so recording synchronization operations will not record the exact order of the write operations constituting the data races. Recording the ordering of operations is only sufficient when there are no data races present.

This precondition is usually not a problem since most data races are a programming error and need to be removed anyway. However, during our adaptation of the Jalapeño code, we found that it contains numerous data races. These data races are no programming errors but are clearly benign. They were put there intentionally to increase performance.

For example, Jalapeño has a “lazy” compilation approach i.e. it will compile a method when it is first executed. Until this happens, the method’s body is implemented by a small stub that will invoke the compiler. This stub will then replace itself with the newly compiled code. There is no synchronization before entering this stub (at the call to the method) since this synchronization would slow down the code even after the method was compiled. This means that multiple threads can enter the stub method and try to start compiling. Only one of them will actually install the compiled code but whether the other threads will still be able to enter the stub method or will already see the newly compiled code is totally timing dependent.

There are other points in Jalapeño which contain data races but they all have in common that the final result of the data race is predetermined. Only the path to get there can vary greatly and can even contain a non-determined number of synchronization operations.

3.2 The deadlock problem

In order to still be able to use the synchronization recording approach, we need to find a way around these benign data races. The most obvious approach is to simply turn off the record/replay infrastructure when we enter a region with a benign data race and turn it on again when we leave the region. Using this approach we ran into a problem when using Lamport clocks as illustrated in Figure 3.

On the left of Figure 3, we see the Lamport clocks as they are recorded during the record phase. Suppose

that the hatched region contains a benign data race. During the execution of this code we turn off the record/replay mechanism so the Lamport clocks aren't updated.

On the right, we try to perform a replay based on the recorded Lamport clocks but run into a deadlock. The deadlock occurs when T_2 is able to obtain the black lock first before T_1 (as in the Figure). In that case, T_1 has to wait till T_2 releases the lock. However, T_2 will continue to run up till (but not including) Lamport clock 6. At this point, the replay mechanism will detect that T_1 still has a pending lock to take with Lamport clock value 5 and therefore T_2 will not be allowed to progress. The final result is that T_1 is waiting for T_2 to release the black lock while T_2 is waiting for T_1 to perform the synchronization with Lamport clock 5 resulting in a deadlock.

The cause for the deadlock lies in the fact that the Lamport clocks introduce additional orderings on top of those actually observed during the record phase. These additional orderings do not take into account the fact that a lock/unlock needs to be performed by T_1 and through coincidence contradict the true behavior of the application.

The solution to this problem lies in not using Lamport clocks but version counters [3]. The idea is simple: every synchronization object is given a version counter initialized to 0. Each time a thread uses a synchronization object the counter is incremented. During the record phase, the consecutive counters seen by each thread are stored into a trace file. Then during replay, a thread will only be allowed to proceed with a synchronization operation if the synchronization object involved has the same version counter as recorded. Since version counters do not introduce any additional ordering, deadlock is avoided.

4 Practical implementation

4.1 Instrumenting data

When implementing the above approach in Jalapeño we chose to add an invisible field to the header of every object. This field can then be used to hold a reference to an instrumentation object of choice. Usually, this field is kept empty. Only when the object is involved in a synchronization operation is it filled with a reference to an object containing the version counter of this object. This additional header field is only visible to special code, called “magic code”, so it does not interfere with normal operation of the code running on top of the virtual machine.

Thread objects are instrumented directly in the Java source code. This removes a level of indirection (loading the reference to the header object) and is therefore beneficial to the execution speed. This is important since thread objects are consulted continuously at every synchronization operation.

4.2 Instrumenting code

In order to intercept every synchronization operation, we've instrumented the compiler. The compiler is responsible for generating code for three synchronization constructs: `monitorenter`, `monitorexit` and `synchronized` methods. The bytecodes `monitorenter` and `monitorexit` are responsible for locking and unlocking an object. When code for these bytecodes is generated, a jump is inserted to the instrumentation code that will update/check the version counters. Similarly, when a `synchronized` method is executed, the compiler will insert code that automatically locks an object at entry and unlocks it again at exit of the method. Here too we insert a call to the instrumentation routines.

The Java class libraries also contain synchronization constructs. A thread can `wait` on an object until another thread calls the `notify(all)` method on that object. In the case of `notify` only one of the threads waiting on an object will be allowed to proceed. It is not specified which thread should be allowed to proceed. This is therefore clearly a non-deterministic decision that should be recorded and replayed. Luckily, before one can call the `wait` or `notify(all)` methods, one has to obtain the lock on that object. Therefore, the order of these operations will be recorded and replayed properly simply by recording the order of the locks.

Another non-deterministic construct we found was the class loading mechanism. When a class is loaded and when it has a static initializer, this initializer will be executed. It is possible that multiple threads will want to load a new class simultaneously. At this point, any thread might execute the initializer code. In order to remove this form of non-determinism, we record which thread initializes which class. When a thread

during replay phase wants to initialize a new class, it is only allowed to proceed if it has initialized the same class during the record phase.

Our record/replay infrastructure itself also loads additional classes depending on whether it is functioning in record or replay mode. In order to make this class loading as invisible as possible, we preload all classes before the main application starts functioning. Of course, the compilation of these classes will result in a varying memory consumption. As a result, the garbage collector will be activated at different points during the execution. Therefore we chose not to try to replay the memory subsystem of the Jalapeño machine but turn off the record/replay infrastructure. Since the memory subsystem is designed to be totally invisible to the main application, we are still able to replay the main application.

4.3 Real-life testing

The proof of the pudding is in the eating. We tried therefore to run SPEC JBB2000. This benchmark simulates a 3-tier system and represents an order processing application for a wholesale supplier. It is highly multi-threaded and non-deterministic. In order to be able to record/replay this real-life benchmark, we needed to add a small additional modification to the Jalapeño machine. The SPEC JBB2000 benchmark requires external input in the form of the current system time. We added a small logging facility that records all the timer readings during the record phase and presents these back to the application during the replay phase. Using these modifications, we were able to record and replay the SPEC JBB2000 benchmark.

5 Conclusions

We developed a record/replay system for Jalapeño capable of replaying applications on a multi-processor. During this development, we found that Jalapeño contains many benign data races which normally would not allow the implementation of record/replay using a synchronization based approach. A major contribution of this work is that we found it is possible to ignore these data races and only replay the main application running on top of Jalapeño while skipping over the data races. To avoid deadlocks with this approach we had to forsake the use of Lamport clocks in favor of version counters.

References

- [1] K. M. R. Audenaert and L. J. Levrouw. Interrupt replay: a debugging method for parallel programs with interrupts. *Microprocessors and Microsystems*, 18(10):601–612, December 1995.
- [2] 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.
- [3] 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.
- [4] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russel, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.
- [5] Jong-Deock Choi, Bowen Alpern, Ton Ngo, Manu Sridharan, and John Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *15th International Parallel and Distributed Processing Symposium*, San Fransisco, April 2001.
- [6] David F. Bacon and Seth Copen Goldstein. Hardware-assisted replay of multiprocessor programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, volume 26, pages 194–206, Santa Cruz, CA, May 1991.
- [7] Luk J. Levrouw and Koenraad M. Audenaert. Reducing the space requirements of instant replay. In *Proceeding ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 205–207. May 1993.

- [8] Luk J. Levrouw and Koenraad M. Audenaert. An efficient record-replay mechanism for shared-memory programs. In *Proceedings Euromicro Workshop on Parallel and Distributed Processing*, pages 169–176. IEEE computer society press, January 1993.
- [9] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.