# JaReC: a record/replay system for multi-threaded Java applications

*Andy Georges, Mark Christiaens, Michiel Ronsse and Koen De Bosschere*

Department ELIS
Ghent University, Sint-Pietersnieuwstraat 41
B-9000 Gent, Belgium
{ageorges, mchristi, ronsse, kdb}@elis.rug.ac.be

**Abstract**

We are focusing on techniques to aid the debugging of multi-threaded Java applications. Debugging such applications is usually quite difficult because of synchronization race conditions. Hence, some special techniques must be applied if one wants to repeat the outcome of the races.
In this paper, we describe such a technique, i.e. performing a controlled replay of a multi-threaded Java application. We eliminate the non-determinism during the replay of the application by imposing order on the synchronization operations executed, as traced during the recorded execution of the application.
Since Java is portable, we want the record/replay system to be at least as portable. Thus, we propose to implement everything in Java, with no or very small changes to the Java Virtual Machine used.

# 1 Introduction

A common technique used to debug a program is cyclic debugging. During cyclic debugging the programmer repeats the execution of a program over and over again and slowly zooms in on a part of the program where he thinks an error may occur. A prerequisite for cyclic debugging is clearly that one must be able to reproduce the erroneous execution at will.
In multi-threaded programs, the cyclic debugging technique often runs into problems due to the presence of race conditions. Simply put, a race condition occurs when two or more threads can simultaneously manipulate a common data structure in an order that is not imposed by the executing program. Consequently, the result of the program execution becomes non-deterministic and therefore a bug may only appear intermittently depending on the outcome of the race. This makes cyclic debugging very hard, since the required repeatability is lost.
Our aim is to eliminate this non-determinism during debugging by forcing the program to behave identically each time it is executed. This way, one can use traditional cyclic debugging techniques to check the program.
To reach this goal we have implemented a record/replay system called JaReC (Java record/replay). Record/replay consists of two phases. During the record phase, the behaviour of the program is traced with as little intrusion as possible. During the replay phase, the main goal is to do a faithful replay of the original run, allowing us to employ highly intrusive debugging techniques without altering the execution of the program.
The record/replay technique used in JaReC is based on recording and replaying the order of synchronization operations performed by the application. Since usually the fraction of synchronization operations is fairly small compared to the total number of operations, JaReC can record these synchronization operations with fairly little overhead.
Our approach is extremely portable and can be used with little or no modifications to the underlying Java Virtual Machine (JVM)[10], whereas most existing replay systems for Java need to extensively modify the internals of the JVM [3, 4, 12].
The rest of this paper is organized as follows. In Section 2 we briefly discuss two primary causes of non-determinism in multi-threaded programs. We then focus on the record and the replay phase in Section 3. We show how we can record the non-deterministic events occurring during the execution and

how we can enforce the same order during the replay phase. In Section 4 we show how the instrumentation of the Java classes is done. We end with some measurements obtained using our record/replay implementation and a brief discussion of related work.

# 2 Non-determinism due to races

In a multi-threaded program it is common that threads share data. If two (or more) threads can access such a shared data structure in a non-synchronized manner, where at least one of the threads changes the data structure, we say that a race condition occurs [1]. The outcome of a race is timing dependent and hence a source of non-determinism.
Data races occur when there is insufficient synchronization present to avoid the interleaving of operations of at least two threads on a common data structure. The threads can be said to 'race' to manipulate the object.
To avoid this behaviour, one usually adds sufficient synchronization to guide the interaction between the threads. Many synchronization constructs have been devised to achieve this aim such as monitors, semaphores, mutexes, condition variables, ...
But adding synchronization introduces so called 'synchronization races'. This is illustrated in Figure 1. We see two threads $T_1$ and $T_2$ using a lock to synchronize their manipulations of the shared object s. Only one thread can acquire the lock (indicated by an opening bracket). The other threads must wait if they want to acquire the lock until it has been released (indicated by a closing bracket) by the thread holding it.
As we can see the execution still is not deterministic. Instead of racing to manipulate s, $T_1$ and $T_2$ are now racing to obtain the lock.
While it is normal for synchronization races to occur, since the non-determinism introduced by synchronization is usually a useful feature of a parallel program, they pose a problem when one wants to debug that program. It is not enough to provide the program execution with the same input to obtain identical results, as would be the case in a single threaded program.
Our technique fixates the non-determinism due to synchronization races, making it possible to repeat an execution faithfully.

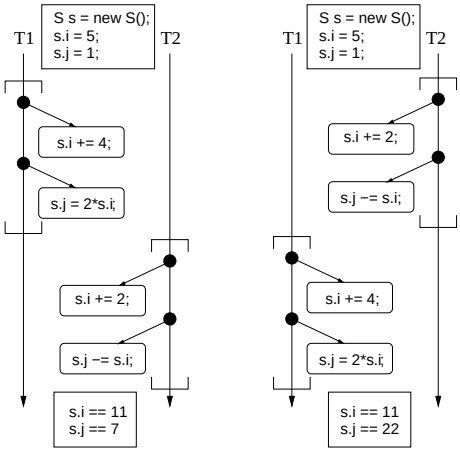


Figure 1: A synchronization race

# 3 Record/replay in Java

The record/replay system we propose to implement is a so called 'ordering' based record/replay system. This means that we trace the order of certain events in the program execution.
It is unfeasible to trace the exact order of the instructions executed by the various threads in the program execution, not only because of the huge amount of trace data this generates, but also because of the fact that the JVM does not impose a total ordering of the operations executed by the threads in the execution.
To avoid these problems, we opt for the approach where only the order of the synchronization operations will be traced during the record phase. During the replay phase, this order will be imposed again producing a faithful replay of the original execution.

## 3.1 Synchronizations in Java

The main synchronization construct in Java is a 'monitor'. It suffices to record the use of the monitors to be able to perform a faithful replay of an application since other synchronization constructs (like starting threads,

waiting for threads, signals, ...) all are built on top of the monitor construct. A monitor in Java consists of a block of statements guarded by a lock on an object. Entering the monitor means that the lock on the associated object is taken. Leaving or exiting the monitor means the lock is released. The same object may be used to guard multiple monitors. Only one thread may hold a lock on an object. Hence, other threads must wait before the monitor until the lock has been released.
In Java, a monitor can be expressed in three different ways.

1. In Java, the lock on an object O can be explicitly taken by using the `synchronized(O){block}` statement. This statement is translated into Java bytecodes (`monitorenter` and `monitorexit`) to resp. obtain the lock on O and to release it.

2. A member function can have a `synchronized` attribute. In this case a lock is acquired by the thread on the `this` object, i.e. the object whose member function was called. No specific instructions to obtain and release the lock are included in the bytecode for the method. Instead, the JVM will check the attributes for the method it will execute. If the method is synchronized, the JVM code itself takes a lock on the `this` object.

3. A static member function can also have the `synchronized` attribute. Here the thread acquires the lock on the object (of `java.lang.Class`) representing the class whose member function was invoked. Here as well, no special bytecode instructions are generated during the compilation of the Java class.

In Java, threads can wait until they receive a notification to proceed. Since threads can only wait on an object for which they hold a lock, the wait will cause the lock to be released. Upon notification the thread will then compete to obtain the lock once more.
If one wants to execute a multi-threaded application, there must be a way to begin executing threads and to wait until they finish their given task. In Java this is implemented by the `start()` and `join()` methods of the class `java.lang.Thread`.

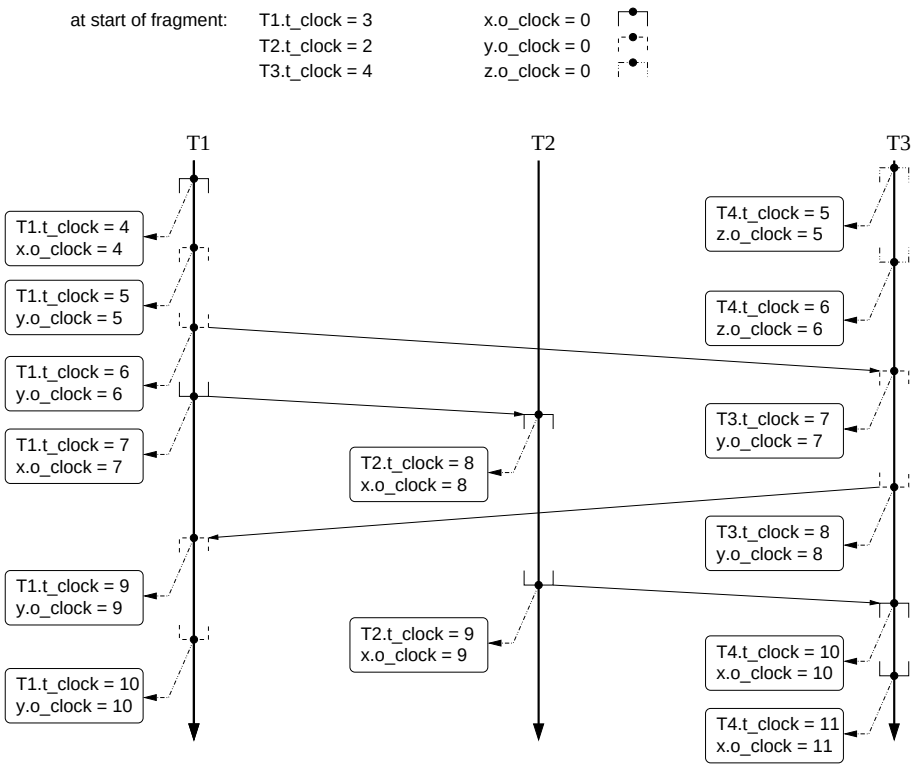


Figure 2: Using Lamport clocks to identify the order of events

## 3.2 The recording phase

When doing ordering based record/replay, we must record the order in which threads perform synchronization operations. We do this by assigning a 'Lamport clock' [9] to every thread. A Lamport clock is nothing more than an integer value. Every time a thread performs a synchronization, we record the 'Lamport clock time' value of this synchronization and update the clock.
The updating goes as follows. When thread $T_i$ executes a synchronization operation with associated object O, we compute the new Lamport clock values of both $T_i$ and O:

$$\mathtt{LC}_{\mathtt{new}} = \max\left(\mathtt{LC}_{T_i}, \mathtt{LC}_O\right) + 1. \tag{1}$$

Then the clock of both the thread and the object are set to the newly calculated value:

$$\mathtt{LC}_{T_i} = \mathtt{LC}_{\mathtt{new}} \quad \text{and} \quad \mathtt{LC}_{\mathrm{O}} = \mathtt{LC}_{\mathtt{new}}. \tag{2}$$

In this way, the object O is used to transmit the Lamport clock time of the last synchronization operation involving O from one thread to another.
To implement this scheme, we assign to each object an 'object clock', in which we store the Lamport clock timestamps $\mathtt{LC}_O$. On top of that, we also store the 'thread clock' values $\mathtt{LC}_{T_i}$ in the Thread objects. Thread objects also have a buffer in which consecutive Lamport clock timestamps can be stored until they are flushed to permanent storage. In this buffer only 'thread clock' values are kept.
An example of a record phase is shown in Figure 2, where three threads have already been started by the 'main' thread (the thread that executes the `main` method of the application). Their clocks have already received a value before the start of the fragment shown. The monitor operations of entering and exiting a monitor are given by the brackets opening to the bottom, resp. the top. The arrows between monitor operations indicate a dependency, meaning that the source of the arrow must precede its target. In each synchronization operation both the 'thread clock' of the thread executing the synchronization operation and the 'object clock' of the object associated with the synchronization operation are adjusted as given in the formulas 1 and 2.
When a thread $T_i$ is started by a thread $T_j$, it is paramount that the 'thread clock' value of $T_j$ is stored in the 'thread clock' field of thread $T_i$, but it

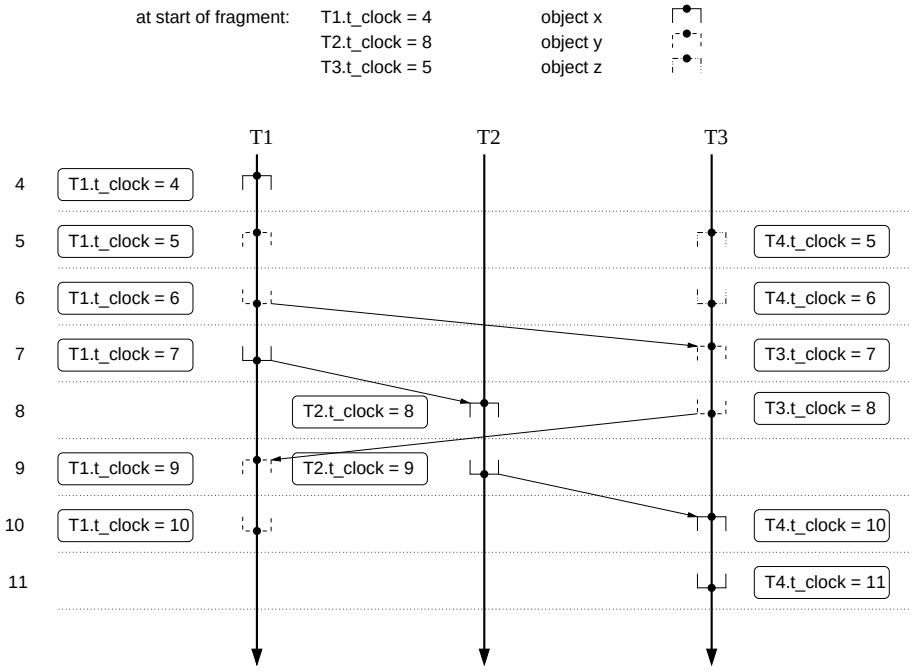


Figure 3: Using the recorded clocks to replay the execution

must not be flushed to the trace, since $T_i$ did not perform a synchronization operation to obtain this time stamp. This way, we are certain that the first synchronization operation executed by $T_i$ will be ordered after the synchronization operations performed by $T_j$ prior to starting $T_i$. Similarly, when two threads join, the 'thread clock' value of the thread ceasing its execution must be transmitted back to the thread waiting for the join to happen.

## 3.3 The replay phase

During this phase we make sure that the synchronization operations executed by all threads are done in the same order as during the record phase. We have a trace file containing a number of Lamport clock values for each thread. With each clock value corresponds a synchronization operation executed during the record phase. When a thread T executes a synchronization

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| T1 | 4 | 4 | 5 | 6 | 7 | 9 | 9 | 10 | $\infty$ |
| T2 | 8 | 8 | 8 | 8 | 8 | 8 | 9 | $\infty$ | $\infty$ |
| T3 | 5 | 5 | 6 | 7 | 8 | 10 | 10 | 10 | $\infty$ |

Figure 4: Clock values evolution during replay

operation, we assign to T the next clock value from the clock values we stored in the trace for T.
Each time a thread T desires to execute a synchronization operation, we check if there is another thread $T_i$ with a smaller clock value than the clock T has at that moment. If this is the case then T has to wait. Otherwise T can continue it's execution. If T has to wait, it will be woken when its clock becomes the smallest clock value of all living threads present. If threads T has the same clock as the threads $T_{j_1}, \ldots, T_{j_k}$, then $T, T_{j_1}, \ldots, T_{j_k}$ were executing concurrently during the record phase and may do so now as well.
The replay phase for the example from Figure 2 is given in Figure 3. The evolution of the clock values assigned from the trace to each thread is shown in Figure 4. A new column in the table indicates that a synchronization operation has been executed.
We can see that the only threads allowed to continue their execution are those with the smallest Lamport clock value.

# 4 Adding instrumentation code

One of the most important features of our record/replay system is the fact that the bulk of the system is implemented in Java. As a consequence our implementation is extremely portable. The system relies on the instrumentation of the Java byte code which will be executed by the JVM. Since a JVM may retrieve class files from a server somewhere on a network, it is not always possible to instrument class files before the JVM decides to load them. Hence, we must add the instrumentation code on the fly, while the JVM is running.
Another reason not to statically instrument class files is the fact that this would require two versions of each class file: an instrumented one and a non-

instrumented one. Furthermore we want the record/replay to be a property of an execution instead of a property of an application. Therefore we want to add the instrumentation at run-time.
Since the JVM must not notice that we are meddling inside the class files, we need to add the instrumentation after the class file has been read from disk, but before it has been loaded into the virtual machine. Some JVMs can use the JVMPI to obtain this goal, in others some code must be added to transfer the class file to the instrumenting Virtual Machine.
To manipulate the class byte code on the instrumenting JVM, we use the BCEL (ByteCode Engineering Library) [7].
When we receive a class on the server[1] side, we change a number of things to prepare the class for record and replay.

## 4.1 Instrumenting the Thread class

Essential to our record/replay system is the instrumentation of the `Thread` class. A thread is the entity that executes synchronization operations. Since a Lamport clock value is assigned to each synchronization operation, a thread must be able to keep track of this clock. As each thread only keeps track of its own Lamport clock, we need to add a field in which we can store the Lamport clock value assigned to the last synchronization. We need this value to calculate the clock value assigned to the next synchronization operation as shown in the formulas 1 and 2. This field is *not* the same as the field given to each object used to pass clock values from one thread to another.
Since it would be far too intrusive to store the acquired clock value in the trace file each time a synchronization operation is executed, we will give each thread an field where we can store the acquired Lamport clock values.
Furthermore, we need a way to uniquely identify a `Thread` object, both during the record and the replay phase. Each `Thread` object instantiated must have the same identification during the replay phase as it had while recording the execution, since we must be able to assign the clock values from the trace to the correct thread when replaying.
Each time a thread is created we force the creating thread to synchronize. This synchronization operation is then logged to the trace file. Since all

[1] We call the instrumenting JVM the server, and the JVM doing the record/replay the client.

threads synchronize on the same object when spawning a new thread, the creation order will be re-playable.

## 4.2 Instrumenting the objects used in the Virtual Machine

As each object O can be associated with a synchronization operation, we must be able to keep track of the Lamport clock value assigned to that object, in order to transmit the clock value of the previous thread performing a synchronization with associated object O.
In Java, all objects are ultimately derived from `java.lang.Object`, so an easy method to give each object instantiated by the JVM a Lamport clock would be to add a field to the `java.lang.Object` class. However, this is in general not allowed by the JVM. Some classes are hard coded into the JVM. This means that changing the Java representation of a class may cause the loss of correspondence between the data structures internally used by the JVM and those visible on the Java level, causing the virtual machine to crash.
Since changing the class `Object` is not feasible, we will add a clock field to each class we instrument. Naturally, we must make sure the clock field is initialized each time an object is constructed. It is also important to make sure we don't inadvertently set the clock field of a `super` object. To solve this last problem, we can add a `getClock()` and a `setClock(int)` method to each class; calling this function will set or get the value of the clock field of the object itself, not of a `super` object.

## 4.3 Instrumenting the `monitorenter` and `monitorexit` instruction

We replace every `monitorenter` and `monitorexit` instruction by a sequence of instructions as given in Figure 5. We make sure that the same instrumented version can be used for both record and replay. In the record phase, we simply increase the clock, while during the replay phase, we also make sure a thread does not execute past a synchronization operation if it does not have the right clock value. Also, during replay, the original synchronization operation is no longer executed. This means that no `wait()` or `notify()`

```
1. getstatic RecPlayMode.isReplay
2. ifeq 5
3. invokestatic
     RecPlayManager.manageThread
4. goto 7
5. dup
6. monitorenter
7. invokestatic
     RecPlayManager.increaseClock
```

(a)

```
1. getstatic RecPlayMode.isReplay
2. ifeq 6
3. invokestatic
     RecPlayManager.manageThread
4. invokestatic
     RecPlayManager.increaseClock
5. goto 9
6. dup
7. invokestatic
     RecPlayManager.increaseClock
8. monitorexit
9. <next instruction>
```

(b)

Figure 5: Instrumentation of the `monitorenter` (a) and `monitorexit` (b) instruction.

```
1.  dup
2.  getstatic RecPlayMode.isReplay
3.  ifeq 12
4.  invokestatic
      RecPlayManager.manageThread
5.  invokestatic
      RecPlayManager.increaseClock
6.  dup
7.  monitorexit
8.  invokestatic
      RecPlayManager.manageThread
9.  dup
10. monitorenter
11. goto 14
12. invokestatic
      RecPlayManager.increaseClock
13. wait
14. invokestatic
      RecPlayManager.increaseClock
```

Figure 6: Instrumentation of a wait instruction.

can be executed, since these methods require that the thread executing them has a lock on the object it is waiting on or it is notifying.

## 4.4 Instrumenting the invocation of `wait()`, `notify()` and `notifyAll()`

Essentially, a `wait()` is a sequence where the thread releases the lock it holds, pauses its execution until it receives a notification and reacquires the same lock.
Obviously we need to trace these synchronization operations as well. The adjustments made to the bytecode are shown in Figure 6. As we can see, during the replay phase, the actual invocation of the `wait()` is no longer executed. This was also the requirement mentioned in section 4.3.

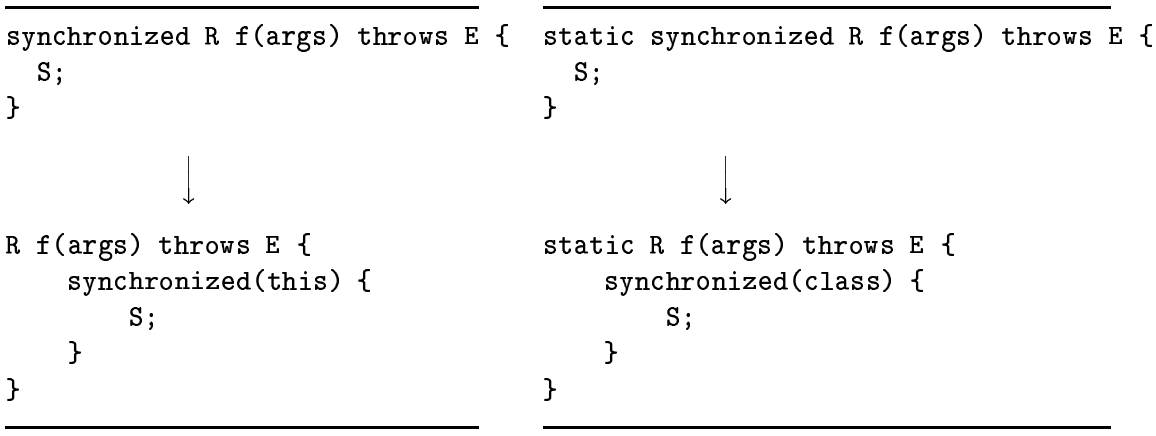


Figure 7: Replacing synchronized member functions.

## 4.5 Instrumenting synchronized methods

A thread must not execute a synchronization operation if it won't be allowed to do so by the replay system. Hence, it must wait *before* the synchronization operation. This can cause problems in the case of (static) synchronized methods.

To solve this problem, we replace each synchronized method by an equivalent method with an explicit `synchronized(Object) { block }`. In this way, all synchronizations are visible in the byte code (except for the wait operation, which we will discuss further on): they are represented by the `monitorenter` and `monitorexit` instructions. Replacing the methods is done as shown in Figure 7.

Of course, exceptions thrown in the method must be caught and the monitor must be exited in a correct manner. To achieve this, we use a finally clause, where we explicitly exit the monitor, thereby releasing the lock we acquired when entering the monitor.

To get the `class` object for the instrumented static synchronized method, we use the `forName(String)` method found in `java.lang.Class`.

| | |
|---|---|
| DinPhil | multi-threaded 'Dining Philosophers' application |
| Barrier | Java Grande Forum BarrierBench |
| 202_jess | jvm98 spec benchmark |
| 202_db | jvm98 spec benchmark |
| Raja | a multi-threaded ray tracer |

Table 1: Benchmarks used to measure the performance of JaReC

### 4.6 Instrumenting the `main` method

Some virtual machines (e.g. the Sun JVM) have threads up and running before the actual *main* thread $T_M$ is constructed and started. Others have no threads running before $T_M$ starts executing the `main` method. To ensure that we are in a well defined state at the start of the application, we must add some initialization code to the `main` method.
The 'thread' and 'object clock' of $T_M$ must be initialized to 0, the clock buffer should be empty, .... All this must be done before the first instruction from the original program is executed. At this point we can also set up the connection to the machine collecting or transmitting the Lamport clock values. Furthermore, since more than one `main` method can be present in a Java application, we must make sure that the initialization happens only once.

## 5 Evaluation results

We have evaluated our record/replay system on a number of benchmarks, a list of which is given in Table 1. Our measurements are given in figures 8 to 11. The times measured here have been obtained by using the SUN JDK 1.2.2, running on an AMD Athlon 1GHz with 1GB of RAM. All Java Virtual Machines were running on the same host, i.e. the JVM doing the instrumentation, and the one storing and sending the clock values.
To speed up execution and to cause less intrusion, especially during the recording phase, we have added an instrumented file cache, in which instrumented class files $C^i_j$ are stored together with the original versions $C^o_j$. If a class $C^o$ can not be found among the $C^o_j$ files, it is instrumented, giving $C^i$ and

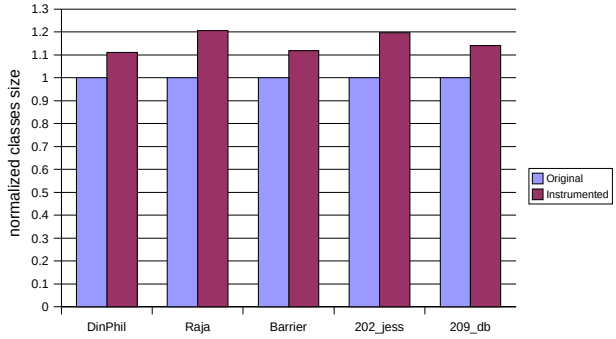


Figure 8: Total class size increase due to instrumentation

added to the cache. During the first run this cache is obviously empty. During subsequent runs however, there is no more need to instrument the class files again, since all instrumented classes can now be fetched immediately from the cache.

We have used one benchmark from the Java Grande Forum thread benchmark. In this benchmark some dependencies on the time were removed, to assure it could be executed using our record/replay system. It has also been made data race free with TRaDe [6].

As we can see in figure 8, the instrumentation increases the size of the class files with about 10 to 25 percent. This includes not only the byte code instructions, but the additions to the constant pool as well. Obviously the increase in instructions depends on the number of `monitorenter` and `monitorexit` instructions as well as the number of instructions to invoke a `wait` and `notify` method, ... in the program.

To have an idea of the overhead involved when doing record/replay with our implementation, we have measured the total execution time and the execution time of the application itself, meaning we checked the time it took from the first instruction of the `main` method to be executed up to the exit of the last non-daemon thread running. This does not include the writing of the clocks during the record phase or the reading of the clocks during the replay phase. We feel this number gives a precise idea of the overhead involved while running the application.

The Barrier benchmark does little else but synchronizing, while the DinPhil

benchmark does a lot of waiting in a random way. The other benchmarks are doing useful work in between synchronizations. From the results for Barrier we see that the overhead caused by the record is about 500 percent. The overhead during replay is worse, since threads have to continuously wait until they are notified, in which case they can execute a synchronization operation, and then they must wait once more and so on. Obviously this a a pathological case, but it shows that, while quite slow, the replay system does work correctly.
For the Raja benchmark the high overhead is clearly caused by the fact that we're running the JVMPI. Our measurements show that the additional burden of performing a record in this case is in fact negligible.
We can conclude that during the record phase, if we look at the slowdown for each synchronization operation, the execution is slowed down about five times (Barrier benchmark). While this seems to be a big slowdown, in a normal application the percentage of synchronization operations will be significantly lower.
Since the amount of intrusion is thus determined by the number of synchronization operations compared to the total number of operations executed, for applications doing useful work other than synchronizing threads, the slowdown will be much lower.

# 6 Related work

As performing a controlled replay of an application execution is a commonly accepted technique to aid in the debugging of a multi-threaded application, several approaches have been developed.
One approach is used by Choi and Srinivasan [4], where they trace the *logical* thread schedule. The authors do not only record the synchronization operations, such as `monitorenter`, `monitorexit`, `wait()` and `notify()/notifyAll()`, but also the accesses to shared variables, which they call critical events. Note however that they do not trace *all* critical events, but instead identify critical event intervals. For this they make use of both global and local clocks (i.e. time stamps). The overhead in execution speed incurred by this method, is less than a 100% for all the benchmarks they used, both for record and for replay. However, this speed is obtained by modifying the Java Virtual Machine itself, something we did not wish to do. Konuru, Srinivasan and

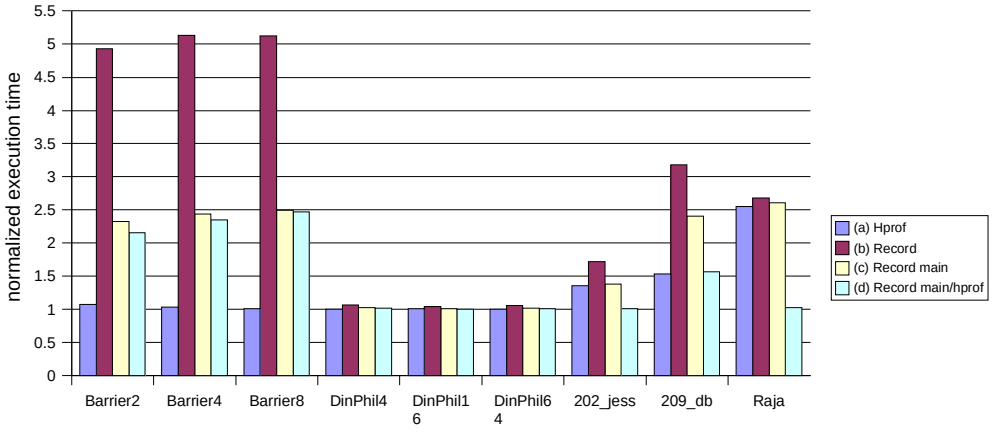


Figure 9: Execution times for the record phase: from left to right the bars represent the normalized time for (a) a normal, not instrumented execution with the JVMPI turned on but doing nothing; (b) the record phase, total execution time; (c) the record phase, without the time needed to store clock values; (d) the record phase (c), normalized with respect to (a).

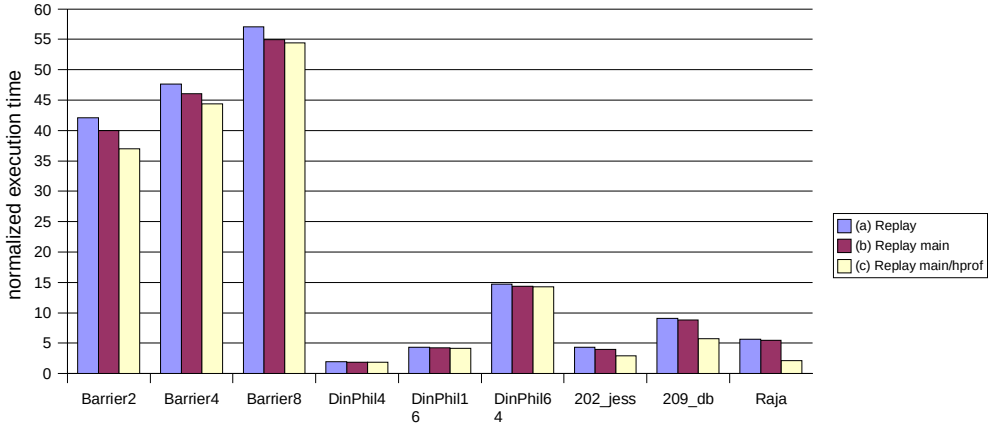


Figure 10: Execution times for the replay phase: from left to right the bars represent the normalized time for (a) the replay phase, total execution time; (b) the replay phase, without the reading of the clock from disk, but with the transmission of the clocks included; (c) the replay phase (b), normalized with respect to the (a) values in Figure 9.

Choi have also adapted their DejaVu record/replay implementation to handle deterministic replay of distributed Java applications [8].
Our approach is similar to the one used by Ronsse and De Bosschere in RecPlay [11]. They have build a record/replay system for a shared memory machine and have implemented their method for Solaris. While their record/replay system has a lot of merits, e.g. it is very fast, it is also tailored to specific platforms and porting it takes a reasonably high effort.
Besides replaying the thread interactions, it is also possible to capture the input of an application and do input record/replay. For Java, such a record/replay system has been implemented by Steven, Chandra et al. [12]. jRapture will record calls to the Java API methods that interact with the underlying (operating) system. During replay jRapture reads the input from the trace and forces it into the Java application.

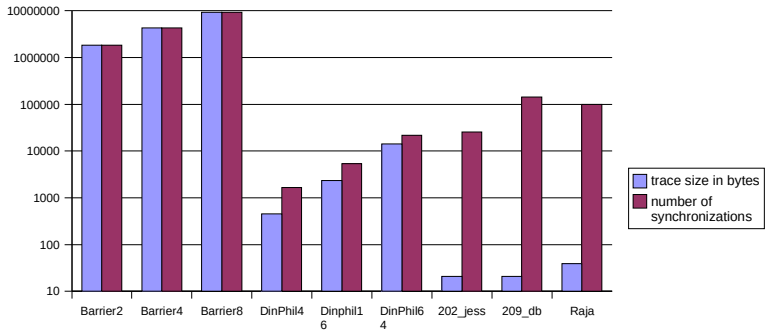


Figure 11: Trace size (in bytes) and number of synchronizations on a logarithmic scale.

# 7 Conclusion and future work

We have built a tool to replay the order of synchronization operations performed by a multi-threaded Java application. As our record/replay system in written in Java, it is extremely portable and can be used by any JVM, provided that the JVM can transmit class files and receive the instrumented versions prior to loading the class into the Virtual Machine. The changes required to the JVM vary from adding some extra functionality to the JVMPI, or adding some communication code inside the JVM. We believe this portability is one of the great merits of JaReC.

The record/replay system facilitates the use of a cyclic debugging method, since the interactions between threads are done in the same order as in the record phase. The total amount of overhead introduced during the record phase is highly dependent on the amount of synchronization operations executed in comparison with 'real' work. We can say that each synchronization operation takes about 5 times as long as in an not instrumented run.

It must be noticed that we record all synchronization operations. This means that even when there is only a single thread executing, we will still introduce the record overhead for each synchronization operation. Moreover, if objects are local to a thread, we need not record the synchronization operations performed on these objects. But if we want to do this, we would need to introduce some form of escape analysis [2, 5, 13] in the system, which was

not our main concern.
Of course, synchronization races are not the only cause of non-determinism in a program. Non-determinism can also be caused by input fed to the application. With JaReC, we have paid no attention to this, we are only interested in replaying application that have repeatable input. Hence, applications using time information cannot be replayed using JaReC. Data race conditions are not our concern either with this record/replay system. We expect the program to be free of data race conditions prior to using JaReC.

# References


[1] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on electronic computers*, 15(5):757–763, October 1966.

[2] B. Blanchet. Escape analysis for object oriented languages: Application to java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA99)*, pages 20–34, Denver, Colorado, 11 1999.

[3] 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.

[4] 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.

[5] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugraban C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA99)*, Denver, Colorado, 11 1999.

[6] Mark Christiaens and Koen De Bosschere. TRaDe, a topological approach to on-the-fly race detection in Java programs. In *Proceedings of the Java Virtual Machine Research and Technology Symposium 2001*, pages 105–116, Monterey, California, USA, April 2001. USENIX.

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

[8] R. Konuru, H. Srinivasan, and J. Choi. Deterministic replay of distributed java applications. In *Proceedings of the 14th IEEE International Parallel & Distributed Processing Symposium*, pages 219–228, May 2000.

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

[10] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 2 edition, April 1999.

[11] 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.

[12] John Steven, Pravir Chandra, Bob Fleck, and Andy 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.

[13] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA99)*, Denver, Colorado, 11 1999.