Ditto – Deterministic Execution Replayability-as-a-Service for Java VM on Multiprocessors

João M. Silva[1], José Simão[2,3], and Luís Veiga[1,2]

[1] Instituto Superior Técnico - ULisboa
[2] INESC-ID Lisboa
[3] Instituto Superior de Engenharia de Lisboa (ISEL)
joao.m.silva@ist.utl.pt, jsimao@cc.isel.ipl.pt, luis.veiga@inesc-id.pt

Abstract. Alongside the rise of multi-processor machines, concurrent programming models have grown to near ubiquity. Programs built on these models are prone to bugs with rare pre-conditions, arising from unanticipated interactions between parallel tasks. Replayers can be efficient on uni-processor machines, but struggle with unreasonable overhead on multi-processors, both concerning slowdown of the execution time and size of the replay log. We present Ditto, a deterministic replayer for concurrent JVM applications executed on multi-processor machines, using both state-of-the-art and novel techniques. The main contribution of Ditto is a novel pair of recording and replaying algorithms that: (a) serialize memory accesses at the instance field level, (b) employ partial transitive reduction and program-order pruning on-the-fly, (c) take advantage of TLO static analysis, escape analysis and JVM compiler optimizations to identify thread-local accesses, and (d) take advantage of a lightweight checkpoint mechanism to avoid large logs in long running applications with fine granularity interactions, and for faster replay to any point in execution. The results show that Ditto out-performs previous deterministic replayers targeted at Java programs.

Keywords: Deterministic Replay, Concurrency, Debugging, JVM.

1 Introduction

The transition to the new concurrent paradigm of programming has not been the easiest, as developers struggle to visualize all possible interleavings of parallel tasks that interact through shared memory. Concurrent programs are harder to build than their sequential counterparts, but they are arguably even more challenging to debug. The difficulty in anticipating all possible interactions between parallel threads makes these programs especially prone to the appearance of bugs triggered by rare pre-conditions, capable of evading detection for long periods. Moreover, the debugging methodologies developed over the years for sequential programs fall short when applied to concurrent ones. Cyclic debugging, arguably the most common methodology, depends on repeated bug reproduction

D. Eyers and K. Schwan (Eds.): Middleware 2013, LNCS 8275, pp. 405–424, 2013.

to find its cause, requiring the fault to be deterministic given the same input. The inherent memory non-determinism of concurrent programs breaks this assumption of fault-determinism, rendering cycling debugging inefficient, as most time and resources are taken up by bug reproduction attempts [1]. Furthermore, any trace statements, added to the program in an effort to learn more about the problem, can actually contribute further to the fault's evasiveness. Hence, cyclic debugging becomes even less efficient in the best case, and ineffective in the worst.

Memory non-determinism, inherent to concurrent programs, results from the occurrence of data races, i.e., unsynchronized accesses to the same shared memory location in which at least one is a write operation. The outcomes of these races must be reproduced in order to perform a correct execution replay. In uniprocessors, these outcomes can be derived from the outcomes of a much smaller subset of races, the synchronization races, used in synchronization primitives to allow threads to compete for access to shared resources. Efficient deterministic replayers have been developed taking advantage of this observation [2–5].

Replaying executions on multi-processors is much more challenging, because the outcomes to synchronization races are no longer enough to derive the outcomes to all data races. The reason is that while parallelism in uniprocessors is an abstraction provided by the task scheduler, in multi-processor machines it has a physical significance. In fact, knowing the task scheduling decisions [6, 7] does not allow us to resolve races between threads concurrently executing in different processors. Deterministic replayers have difficulties with unreasonable overhead when applied in this context, as the instructions that can lead to data races make up a significant amount of the instructions executed by a typical application. Currently there are four distinct approaches to deal with this open research problem, discussed in Section 2. Even using techniques to prune the events of interest, long running applications can make the log of events grow to an unmanageable size. To avoid this, a checkpointing mechanism can also be used to transparently save the state of the program, with the events before the checkpoint truncated from the log. The last saved state, may be potentially smaller than the original untruncated log, and can also be used as a starting point for a future replay allowing for a faster replay solution.

In this paper, we present Ditto, our deterministic replayer for unmodified user-level applications executed by the JVM on multi-processor machines. It integrates state-of-the-art and novel techniques to improve upon previous work. The main contributions that make Ditto unique are: (a) A novel pair of logical clock-based [8] recording and replaying algorithms. This allows us to leverage the semantic differences between load and store memory accesses to reduce trace data and maximize replay-time concurrency. Furthermore, we serialize memory accesses at the finest possible granularity, distinguishing between instance fields and array indexes; (b) Reduced trace and log space. We use a constraint pruning algorithm based on program order and partial transitive reduction to reduce the amount of trace data on-the-fly and a checkpointing mechanism to employ in long running applications; (c) A trace file optimization that highly reduces the size of

logical clock-based traces; Though we discuss and implement Ditto in the context of a JVM runtime, its underlying techniques may be directly applied to other high-level, object-oriented runtime platforms, such as the Common Language Runtime (CLR).

We implemented Ditto on top of the open-source JVM implementation Jikes RVM (Research Virtual Machine). Ditto is evaluated to assess its replay correctness, bug reproduction capabilities and performance. Experimental results show that Ditto consistently out-performs previous state-of-the-art deterministic replayers targeted at Java programs in terms of record-time overhead, trace file size and replay-time overhead. It does so across multiple axes of application properties, namely number of threads, number of processors, load to store ratio, number of memory accesses, number of fields per shared object, and number of shared objects.

The rest of the paper is organized as follows: Section 2 describes some instances of related work; Section 3 explains the base design and algorithms of Ditto; Section 4 presents fundamental optimizations; Section 5 discusses some implementation related details; Section 6 presents and analyzes evaluation results; and Section 7 concludes the paper and offers our thoughts on the directions of future work.

2 Related Work

Deterministic replayers for multi-processor executions can be divided into four categories in terms of the approach taken to tackle the problem of excessive overhead. Some systems replay solely synchronization races, thus guaranteeing a correct replay up until the occurrence of a data race. RecPlay [3] and JaRec [4] are two similar systems that use logical clock-based recording algorithms to trace a partial ordering over all synchronization operations. RecPlay is capable of detecting data races during replay. Nonetheless, we believe the assumption that programs are perfectly synchronized severely limits the effectiveness of these solutions as debugging tools in multi-processor environments.

Researchers have developed specialized hardware-based solutions. FDR [9] extends the cache coherence protocol to propagate causality information and generate an ordering over memory accesses. DeLorean [10] forces processors to execute instructions in chunks that are only committed if they do not conflict with other chunks in terms of memory accesses. Hence, the order of memory accesses can be derived from the order of chunk commits. Though efficient, these techniques have the drawback of requiring special hardware.

A more recent proposal is to use probabilistic replay techniques that explore the trade-off between recording overhead reduction through partial execution tracing and relaxation of replay guarantees. PRES partially traces executions and performs an offline exploration phase to find an execution that conforms with the partial trace and with user-defined conditions [11]. ODR uses a formula-solver and a partial execution trace to find executions that generate the same output as the original [12]. These techniques show a lot of potential as debugging

tools, but are unable to put an upper limit on how long it takes for a successful replay to be performed, though the problem is minimized by fully recording replay attempts.

LEAP is a relevant Java deterministic replayer that employs static analysis, to identify memory accesses performed on actual thread-shared variables, hence reducing the amount of monitored accesses [13]. Because LEAP recording algorithm associates access vectors to fields, it can not distinguish accesses to the same field of different objects. In workloads where there are many objects of a single type but they are not shared among threads, this will diminish the concurrency of the recording and replaying mechanisms. ORDER [14] is, like Ditto, an object centric recorder. From a design point of view, ORDER misses support for online pruning of events and a checkpoint mechanism for faster replay. Regarding current implementation, the baseline code base (Apache harmony) is now deprecated, while Ditto was developed in a research oriented, yet production-like quality JVM, that is widely supported by the research community.

Deterministic replay can also be used as an efficient means for a fault-tolerant system to maintain replicas and recover after experiencing a fault [15, 16].

3 Ditto – System Overview

Ditto must record the outcomes of all data races in order to support reproduction of any execution on multi-processor machines. Data races arise from non-synchronized shared memory accesses in which at least one is a write operation. Thus, to trace outcomes to data races, one must monitor shared memory accesses. The JVM's memory model limits the set of instructions that can manipulate shared memory to three groups: (i) accesses to static fields, (ii) accesses to object fields, and (iii) accesses to array fields of any type.

In addition to shared memory accesses, it is mandatory that we trace the order in which synchronization operations are performed. Though these events have no effect on shared memory, an incorrect ordering can cause the replayer to deadlock when shared memory accesses are performed inside critical sections. They need not, however, be ordered with shared memory accesses. In the JVM, synchronization is supported by synchronized methods, synchronized blocks and synchronization methods, such as `wait` and `notify`. Since all these mechanisms use monitors as their underlying synchronization primitive, their acquisitions are the events that Ditto intercepts. For completeness, we also record values and orderings of external input to threads, such as random numbers and from other library functions, while assuming the content of input from files is available.

3.1 Base Record and Replay Algorithms

The recording and replaying algorithms of Ditto rely on logical clocks (or Lamport clocks) [8], a mechanism designed to capture chronological and causal relationships, consisting of a monotonically increasing software counter. Logical clocks are associated with threads, objects and object fields to identify the order

Algorithm 1. Load wrapper

```
Parameters: f is the field, v is the value loaded
 1: method wrapLoad(f,v)
 2:     monitorEnter(f)
 3:     t ← getCurrentThread()
 4:     trace(f.storeClock)
 5:     f.loadCount ← f.loadCount + 1
 6:     if f.storeClock > t.clock then
 7:         t.clock ← f.storeClock
 8:     end if
 9:     v ← load(f)
10:     monitorExit(f)
11: end method
```

Algorithm 2. Store wrapper

```
Parameters: f is the field, v is the value stored
 1: method wrapStore(f,v)
 2:     monitorEnter(f)
 3:     t ← getCurrentThread()
 4:     trace(f.storeClock, f.loadCount)
 5:     clock ← max(t.clock, f.storeClock) + 1
 6:     f.storeClock ← clock
 7:     f.loadCount ← 0
 8:     t.clock ← clock
 9:     store(f, v)
10:     monitorExit(f)
11: end method
```

between events of interest. For each such event, the recorder generates an order constraint that is later used by the replayer to order the event after past events on which its outcome depends.

Recording: The recorder creates two streams of order constraints per thread – one orders shared memory accesses, while the other orders monitor acquisitions. The recording algorithm for shared memory accesses was designed to take advantage of the semantic differences between load and store memory accesses. To do so, Ditto requires state to be associated with threads and fields. Threads are augmented with one logical clock, the thread's clock, incremented whenever it performs a store operation. Fields are extended with (a) one logical clock, the field's store clock, incremented whenever its value is modified; and (b) a load counter, incremented when the field's value is loaded and reset when it is modified. The manipulation of this state and the load/store operation itself must be performed atomically. Ditto acquires a monitor associated with the field to create a critical section and achieve atomicity. It is important that the monitor is not part of the application's scope, as its usage would interfere with the application and potentially lead to deadlocks.

When a thread T_i performs a load operation on a field f, it starts by acquiring f's associated monitor. Then, it adds an order constraint to the trace consisting of f's store clock, implying that the current operation is to be ordered after the store that wrote f's current value, but specifying no order in relation to other loads. Thread and field state are then updated by incrementing f's load count, and the load operation itself performed. Finally, the monitor of f is released. If T_i instead performs a store operation on f, it still starts by acquiring f's monitor, but follows by tracing an order constraint composed of the field's store clock and load count, implying that this store is to be performed after the store that wrote f's current value and all loads that read said value. Thread and field states are then updated by increasing clocks and resetting f's load count. Finally, the store is performed and the monitor released. Algorithms 1 and 2 list pseudo-code for these recording processes.

Unlike memory accesses, performed on fields, monitor acquisitions are performed on objects. As such, we associate with each object a logical clock. Moreover, given that synchronization is not serialized with memory accesses, we add

Algorithm 3. Recording monitor acquisition operations

```
Parameters: o is the object whose monitor was acquired
1: method afterMonitorEnter(o)
2:     t ← getCurrentThread()
3:     trace(o.syncClock)
4:     clk ← max(t.syncClock, o.syncClock)+1
5:     o.syncClock ← clk
6:     t.syncClock ← clk
7: end method
```

Algorithm 4. Replaying load memory access operations

```
Parameters: f is the field whose value is being loaded into v and is protected by a monitor
1: method wrapLoad(f,v)
2:     t ← getCurrentThread()
3:     clock ← nextLoadConstraint(t)
4:     while f.storeClock < clock do
5:         wait(f)
6:     end while
7:     v ← load(f)
8:     t ← getCurrentThread()
9:     if f.storeClock > t.clock then
10:        t.clock ← f.storeClock
11:    end if
12:    f.loadCount ← f.loadCount + 1
13:    notifyAll(f)
14: end method
```

a second clock to threads. When a thread T_i acquires the monitor of an object o, it performs Algorithm 3. Note that we do not require a monitor this time, as the critical section of o's monitor already protects the update of thread and object state.

Consistent Thread Identification: Ditto's traces are composed of individual streams for each thread. Thus, it is mandatory that we map record-time threads to their replay-time counterparts. Threads can race to create child threads, making typical Java thread identifiers, attributed in a sequential manner, unfit for our purposes. To achieve the desired effect, Ditto wraps thread creation in a critical section and attributes a replay identifier to the child thread. The monitor acquisitions involved are replayed using the same algorithms that handle application-level synchronization, ensuring that replay identifiers remain consistent across executions.

Replaying: As each thread is created, the replayer uses its assigned replay identifier to pull the corresponding stream of order constraints from the trace file. Before a thread executes each event of interest, the replayer is responsible for using the order constraints to guarantee that all events on which its outcome depends have already been performed. The trace does not contain metadata about the events from which it was generated, leaving the user with the responsibility of providing a program that generates the same stream of events of interest as it did at record-time. Ditto nonetheless allows the original program to be modified while maintaining a constant event stream through the use of Java annotations or command-line arguments, an important feature for its usage as a debugging tool.

Replaying Shared Memory Accesses: Using the order constraints in a trace file, the replayer delays load operations until the value read at record-time is available, while store operations are additionally delayed until that value has been read as many times as it was during recording, using the field's load count.

This approach allows for maximum replay concurrency, as each memory access waits solely for those events that it affects and is affected by.

When a thread T_i performs a load operation on a field f, it starts by reading a load order constraint from its trace, extracting a target store clock from it. Until f's store clock equals this target, the thread waits. Upon being notified and positively re-evaluating the conditions for advancement, it is free to perform the actual load operation. After doing so, thread and field states are updated and waiting threads are notified of the changes. Algorithm 4 lists pseudo-code for this process. If T_i was performing a store operation, the process would be the same, but a store order constraint would be loaded instead, from which a target store clock and a target load count would be extracted. The thread would proceed with the store once f's store clock and load count both equaled the respective targets. State would be updated according to the rules used in Algorithm 1. Replaying monitor acquisitions is very similar to replaying load operations, with two differences: (i) a sync order constraint is read from the trace, from which a target sync clock is extracted and used as a condition for advancement; and (ii) thread and object state are updated according to the rules in Algorithm 3.

Notice that during replay there is no longer a need for protecting shared memory accesses with a monitor, as synchronization between threads is now performed by Ditto's wait/notify mechanism. Furthermore, notice that the load counter enables concurrent loads to be replayed in an arbitrary order, hence in parallel and faster, rather than being serialized unnecessarily.

3.2 Wait and Notify Mechanism

During execution replay, threads are often forced to wait until the conditions for advancement related to field or object state hold true. As such, threads that modify the states are given the responsibility of notifying those waiting for changes. Having threads wait and notify on the monitor associated with the field or object they intend to, or have manipulated, as suggested in Algorithms 1-2 and 3, is a simple but sub-optimal approach which notifies threads too often and causes bottlenecks when they attempt to reacquire the monitor. Ditto uses a much more refined approach, in which threads are only notified if the state has reached the conditions for their advancement.

Replay-time states of fields and objects are augmented with a table indexed by three types of keys: (i) load keys, used by load operations to wait for a specific store clock; (ii) store keys, used by store operations to wait for a specific combination of store clock and load count; and (iii) synchronization keys, used by monitor acquisitions to wait for a specific synchronization clock. Let us consider an example to illustrate how these keys are used. When a thread T_i attempts to load the value of a field f but finds f's store clock lower than its target store clock (tc), it creates a load key using the latter. T_i then adds a new entry to f's table using the key as both index and value, and waits on the key. When another thread T_j modifies f's store clock to contain the value tc, it uses a load

key (tc) and a store key ($tc, 0$) to index the table. As a result of using the load key, it will retrieve the object on which T_i is waiting and invokes `notifyAll` on it. Thus, T_i is notified only once its conditions for proceeding are met.

3.3 Lightweight Checkpointing

For long running applications, and especially those with fine-grained thread interactions, the log can grow to a large size. Furthermore, the replay can only be necessary to be done from a certain point in time because the fault is known to occur only at the end of execution. Ditto uses a lightweight checkpointing mechanism [17] to offer two new replay services: (i) replay to most recent point before fault; (ii) replay to any instant M in execution. Checkpoint is done recording each thread stack and reachable objects. In general, the checkpoint size is closely related to the size of live objects, plus the overhead of booking metadata necessary for recovery. While the size of live objects can remain consistent over time, the log size will only grow. Regarding scenario (i), replay starts by recovering from the last checkpoint and continues with the partial truncated log. So, the total recording space is $sizeof(lastCheckpoint) + sizeof(truncatedLog)$ which is still bounded to be smaller than $2 * sizeof(checkpointSize)$, since we trigger checkpointing when the log reaches a size close to the total memory used by objects (90%). In scenario (ii), replay starts with the most recent checkpoint before instant M (chosen by the user), and the partial log collected after that instant. In this case, the total recording space is $N * sizeof(checkpoint) + N * sizeof(truncatedLog)$, where N is the number of times a checkpoint is done. In this case there is a trade-off between overhead in execution time and granularity in available replay start times [17]. Even so, the total recording space is bounded to be smaller than $2 * N * sizeof(checkpoint)$.

3.4 Input Related Non-Deterministic Events

Besides access to shared variables, another source of non-determinism is the input some programs use to progress their calculus. This input can come from information asked to the program's user or from calling non-deterministic services, such as the current time or the random number generator. All such services are either available through the base class library or calls using the Java Native Interface. Each time a call is made to a method that is a source of input non-determinism (e.g. `Random.nextInt`, `System.nanoTime`), the result is saved in association with the current thread. If the load/store is made over a shared field, the replay mechanism will already ensure the same thread interleaving as occurred in the recording phase. Regarding non shared fields, the replay of deterministic information can occur in a different order than the one of the original execution. This is not a problem since the values are affiliated with a thread and are delivered using FIFO order during each thread execution.

4 Additional Optimizations

4.1 Recording Granularity

Ditto records at the finest possible granularity, distinguishing between different fields of individual instances when serializing memory accesses. Previous deterministic replayers for Java programs had taken sub-optimal approaches: (i) DejaVu creates a global-order [2]; (ii) LEAP generates a partial-order that distinguishes between different fields, but not distinct instances [13]; and (iii) JaRec does the exact opposite of LEAP [4]. The finer recording granularity maximizes replay-time concurrency and reduces recording overhead due to lower contention when modifying recorder state. The downside is higher memory consumption associated with field states. If this becomes a problem, Ditto is capable of operating with an object-level granularity.

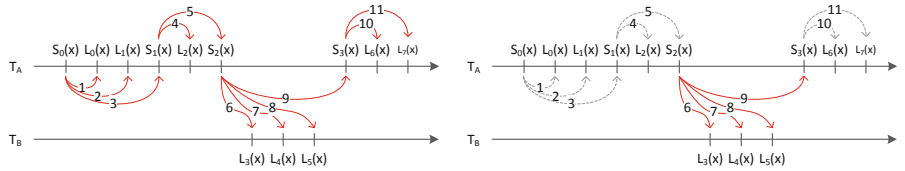
Array indexes are treated like object fields, but with a slight twist. To keep index state under control for large arrays, a user-defined cap is placed on how many index states Ditto can keep for each array. Hence, multiple array indexes may map to a single index state and be treated as one program entity in the eyes of the recorder and replayer. This is not an optimal solution, but it goes towards a compromise with the memory requirements of Ditto.

4.2 Pruning Redundant Order Constraints

The base recording algorithm traces an order constraint per event of interest. Though correct, it can generate unreasonably high amounts of trace data, mostly due to the fact that shared memory accesses can comprise a very significant fraction of the instructions executed by a typical application. Fortunately, many order constraints are redundant, i.e., the order they enforce is already indirectly enforced by other constraints or program order. Such constraints can be safely pruned from the trace without compromising correctness. Ditto uses a pruning algorithm that does so on-the-fly.

Pruning order constraints leaves gaps in the trace which our base replay algorithm is not equipped to deal with. To handle these gaps, we introduce the concept of free runs, which represent a sequence of one or more events of interest that can be performed freely. When performing a free run of size n, the replayer essentially allows n events to occur without concerning itself with the progress of other threads. Free runs are placed in the trace where the events they replace would have been.

Program Order Pruning: Consider the recorded execution in Figure 1(a), in which arrows represent order constraints traced by the base recording algorithm. Notice how all dashed constraints enforce orderings between events which are implied by program order. To prune them, Ditto needs additional state to be associated with fields: the identifier of the last thread to store a value in the field, and a flag signaling whether that value has been loaded by other threads. Potential load order constraints are not traced if the thread loading the value is the



(a) Order constraints traced by base recording algorithm.

(b) Pruning constraints implied by program order.

Fig. 1. Example of Ditto's constraint pruning algorithm

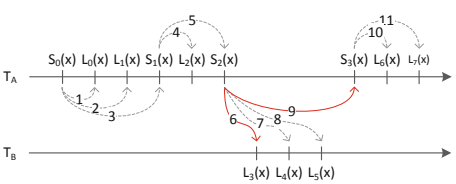


Fig. 2. Pruning constraints implied by previous constraints

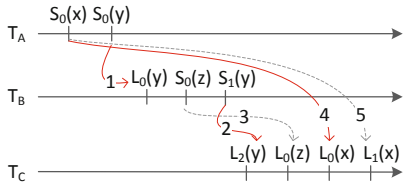


Fig. 3. Example of partial transitive reduction

one that wrote it. Thus, constraints 1, 2, 4, 10 and 11 in Figure 1(a) are pruned, but not constraint 6. Similarly, a potential store order constraint is not traced if it is performed by the thread that wrote the current value and if that value has not been loaded by other threads. Hence, constraints 3 and 5 are pruned, while 9 is not, as presented in Figure 1(b). Synchronization order constraints are handled in the same way as load operations, but state is associated with an object instead of a field.

Partial Transitive Reduction: Netzer introduced an algorithm to find the optimal set of constraints to reproduce an execution [18], which was later improved upon in RTR [19] by introducing artificial constraints that enabled the removal of multiple real ones. Ditto does not directly employ any of these algorithms for reasons related to performance degradation and the need for keeping flexibility-limiting state, such as Netzer's usage of vector clocks, requiring the number of threads to be known a priori. Instead, Ditto uses a novel partial transitive reduction algorithm designed to find a balance between trace file size reduction and additional overhead.

Transitive reduction prunes order constraints that enforce orderings implicitly enforced by other constraints. In Figure 1, for example, T_B performs three consecutive load operations which read the same value of x, written by T_A. Given that the loads are ordered by program order, enforcing the order $S_2(x) \rightarrow L_3(x)$ is enough to guarantee that the following two loads are also subsequent to $S_2(x)$. As such, constraints 7 and 8 are redundant and can be removed, resulting in the final trace file of Figure 2 with only 2 constrains.

To perform transitive reduction, we add a table to the state of threads that tracks the most recent inter-thread interaction with each other thread. Whenever a thread T_i accesses a field f last written to by thread T_j (with $T_i \neq T_j$), f's store clock is inserted in the interaction table of T_i at index T_j. This allows Ditto to declare that order constraints whose source is T_j with a clock lower than the one in the interaction table are redundant, implied by a previous constraint. Figure 3 shows a sample recording that stresses the partial nature of Ditto's transitive reduction, since the set of traced constraints is sub-optimal. Constraint 4 is redundant, as the combination of constraints 1 and 2 would indirectly enforce the order $S_0(x) \rightarrow L_0(x)$. For Ditto to achieve this conclusion, however, the interaction tables of T_B and T_C would have to be merged when tracing constraint 2. The merge operation proved to be too detrimental to efficiency, especially given that the benefit is limited to one order constraint, as the subsequent constraint 5, similar to 4, is pruned. In summary, Ditto is aware of thread interactions that span a maximum of one traced order constraint.

4.3 Thread Local Objects and Array Escape Analysis

Thread Local Objects (TLO) static analysis provides locality information on class fields, that is, it determines fields which are not involved in inter-thread interactions, aiming to save execution time and log space. The output of this kind of analysis is a classification of either thread-local or thread-shared for each class field. We developed a stand-alone application that uses the TLO implementation in the Soot bytecode optimization framework[1] to generate a report file that lists all thread-shared fields of the analyzed application. This file can be fed as optional input to Ditto, which uses the information to avoid intercepting accesses to thread-local fields.

TLO analysis provides very useful information about the locality of class fields, but no information is offered on array fields. Without further measures, we would be required to conservatively monitor all array fields accesses. Ditto uses, at runtime, information collected from the just-in-time compiler to do escape analysis on array references and avoid monitoring accesses to elements of arrays declared in a method whose reference never escapes that same method. This analysis, although simple, can still avoid some useless overhead at little cost. Nonetheless, there is a lot of unexplored potential for this kind of analysis on array references to reduce recording overhead which we see as future work.

4.4 Trace File

Ditto's traces are composed of one order constraint stream per record-time thread. Organizing the trace by thread is advantageous for various reasons. The first is that it is easy to intercept the creation and termination of threads. Intercepting these events is crucial for the management of trace memory buffers, as

[1] `http://www.sable.mcgill.ca/soot/`

they must be created when a thread starts and dumped to disk once it terminates. Moreover, it allows us to place an upper limit on how much memory can be spent on memory buffers, as the number of simultaneously running threads is limited and usually low. Other trace organizations, such as the field-oriented one of LEAP [13], do not benefit from this – the lifetime of a field is the lifetime of the application itself. A stream organized by instance would be even more problematic, as intercepting object creation and collection is not an easy task.

The trace file is organized as a table that maps thread replay identifiers to the corresponding order constraint streams. The table and the streams themselves are organized in a linked list of chunks, as a direct consequence of the need to dump memory buffers to disk as they become full. Though sequential I/O is generally more efficient than random I/O, using multiple sequential files (one per thread) turned out to be less efficient than updating pointers in random file locations as new chunks were added to it. Hence, Ditto creates a single-file trace.

Given that logical clocks are monotonically increasing counters, they are expected to grow to very large values during long running executions. For the trace file, this would mean reserving upwards of 8 bytes to store each clock value. Ditto uses a simple but effective optimization that stores clock values as increments in relation to the last one in the stream, instead of as absolute values. Considering that clocks always move forward and mostly in small increments, the great majority of clock values can be stored in 1 or 2 bytes.

5 Implementation Details

Ditto is implemented in Jikes RVM, a high performance implementation of the JVM written almost entirely in a slightly enhanced Java that provides "magic" methods for low-level operations, such as pointer arithmetic [20]. The RVM is very modular, as it was designed to be a research platform where novel VM ideas could be easily implemented and evaluated. This was the main reason we developed Ditto on it.

The implementation efforts were done in two main sub-systems: threading and compiler. Regarding the threading system, each Java thread is mapped to a single native thread. This is relevant to Ditto, as it means scheduling decisions are offloaded to the OS and cannot be traced or controlled from inside the RVM. As a consequence, Java monitors are also implemented with resort to OS locking primitives. Regarding the compiler, Jikes RVM does not interpret bytecode; all methods are compiled to machine code on-demand. The VM uses an adaptive compilation system in which methods are first compiled by a fast baseline compiler which generates inefficient code. A profiling mechanism detects hot methods at runtime, which are then recompiled by a slower but much better optimizing compiler. This compiler manipulates three intermediate representations (IR) on which different optimizations are performed. The high-level IR (HIR) is very similar to the bytecode instruction set, but subsequent IRs are closer to actual processor ISAs.

Intercepting Events of Interest: Implementing Ditto in Jikes RVM required intercepting the events of interest through hooks in the thread management subsystem and the addition of instrumentation phases to the compilers. Moreover, mechanisms were added to manage thread, object and field states. A drawback of Jikes being written in Java is that it uses the same mechanisms for executing as the application. As such, when intercepting events, we must ignore those triggered by the VM. Depending on the event, the VM/application distinction is done using either static tests that rely on package names, or runtime tests that inspect the Java stack.

Ditto intercepts thread creation, both before and after the launch of the native thread, and thread termination, mainly for the purpose of initializing and dumping trace memory buffers. The thread creation hooks are also used to enter and exit the critical section protecting replay identifier assignment. If the event occurs in the context of a synchronized method or block, Ditto simply replaces the usual method used to implement the monitor enter operation with a wrapper method during compilation. Monitor acquisitions performed in the context of synchronization methods like `wait` or `notify` are intercepted by a hook in the VM's internal implementation of said methods. To avoid costly runtime tests, call sites are instrumented to activate a thread-local flag which lets Ditto know that the next executed synchronization method was invoked by the application. Events triggered through the JNI interface are also intercepted by a hook inside the VM, but they require a runtime test to ascertain the source, as we do not compile native code.

During method compilation, accesses to shared memory are wrapped in two calls to methods that trace the operation. Instrumentation is performed after HIR optimizations have been executed on the method, allowing Ditto to take advantage of those that remove object, array or static field accesses. Such optimizations include common sub-expression elimination and object/array replacement with scalar variables using escape analysis, among others.

Threading and State: Thread state is easily kept in the VM's own thread objects. Object and field states are kept in a state instance whose reference is stored in the object's header. After modifying the GC to scan these references, this approach allows us to create states for objects on-demand and keep them only while the corresponding object stays alive. Ditto requires the trace file to be finalized in order to replay the corresponding execution. When a deadlock occurs, the JVM does not shutdown and the trace memory buffers are never dumped, leaving the trace in an unfinished state. The problem is solved by adding a signal handler to Jikes which intercepts `SIGUSR1` signals and instructs the replay system to finish the trace. The user is responsible for delivering the signal to Jikes before killing its process if a deadlock is thought to have been reached.

Trace File: In Section 4 we described the way thread order constraint streams are located in the trace file using a combination of table and linked list structures. Structuring the streams themselves is another issue, as Ditto's recording algorithm generates three types of values that must be somehow encoded in

the stream: (i) clock increment values; (ii) free run values; and (iii) load count values. Furthermore, the clock value optimization, also presented in Section 4, makes value sizes flexible, requiring the introduction of a way to encode this information as well.

The three kinds of values are encoded using the two most significant bits of each value as identification metadata. However, adding two more bits for size metadata would severely limit the range of values that each entry could represent. Moreover, it is usual for consecutive values to have equal size, leading to a lot of redundant information if the size is declared for each individual entry. Taking these observations in mind, we introduce meta-values to the stream which encode the size and number of the values that follow them in the stream. The meta-values take up two bytes, but their number is insignificant in comparison to the total amount of values stored in the trace. Ditto uses a VM's internal thread whose only purpose is to write trace buffers to disk. By giving each thread two buffers, we allow one buffer to be dumped to disk by the writer thread while the other is concurrently filled. In most cases, writing to disk is faster than filling a second buffer, allowing threads to waste no time waiting for I/O operations.

6 Evaluation

We evaluate Ditto by assessing its ability to correctly replay recorded executions and by measuring its performance in terms of recording overhead, replaying overhead and trace file size. Performance measurements are compared with those of previous approaches, which we implemented in Jikes RVM using the same facilities that support Ditto itself. The implemented replayers are: (a) DejaVu [2], a global-order replayer; (b) JaRec [4], a partial-order, logical clock-based replayer; and (c) LEAP [13], a recent partial-order, access vector-based replayer. We followed their respective publications as closely as possible, introducing modifications when necessary. For instance, DejaVu and JaRec, originally designed to record synchronization races, were extended to deal with all data races, while LEAP's algorithm was extended to compress consecutive accesses to a field by the same thread, absent in available codebase. Moreover, our checkpoint for instant replay is deactivated for fairness.

We start by using a highly non-deterministic microbenchmark and a number of applications from the IBM Concurrency Testing Repository[2] to assess replay correctness. This is followed by a thorough comparison between Ditto's runtime performance characteristics and those of the other implemented replayers. The results are gathered by performing a microbenchmark and recording executions of selected applications (because of space constraints) from the Java Grande and DaCapo benchmark suites. All experiments were conducted on a 8-core 3.40Ghz Intel i7 machine with 12GB of primary memory and running 64-bit Linux 3.2.0. Baseline version of the Jikes RVM is 3.1.2. Ditto's source will be available in the Jikes RVM research archive.

[2] `https://qp.research.ibm.com/concurrency_testing`

Replay Correctness: In the context of Ditto, an execution replay is said to be correct if the shared program state goes through the same transitions as it did during recording, even if thread local state diverges. Other types of deterministic replayers may offer more relaxed fidelity guarantees, as is the case of the probabilistic replayers PRES [11] and ODR [12].

We design a microbenchmark to produce a highly erratic and non-deterministic output, so that we can confirm the correctness of replay with a high degree of assurance. This is accomplished by having threads randomly increment multiple shared counters without any kind of synchronization, and using the final counter values as the output. After a few iterations, the final counter values are completely unpredictable due to the non-atomic nature of the increments. Naively re-executing the benchmark in hopes of getting the same output will prove unsuccessful virtually every time. On the contrary, Ditto is able to reproduce the final counter values every single time, even when stressing the system by using a high number of threads and iterations. The microbenchmark will also be available in the Jikes RVM research archive. Regarding the *IBM concurrency testing repository*, it contains a number of small applications that exhibit various concurrent bug patterns while performing some practical task. Ditto is capable of correctly reproducing each and every one of these bugs.

6.1 Performance Results

After confirming Ditto's capability to correctly replay many kinds of concurrent bug patterns, we set off to evaluate its performance by measuring recording overhead, trace file size and replaying overhead. To put experimental results in perspective, we use the same performance indicators to evaluate the three implemented state-of-the-art deterministic replay techniques for Java programs: DejaVu (Global), JaRec, LEAP.

Microbenchmarking: The same microbenchmark used to assess replay correctness is now used to compare Ditto's performance characteristics with those of the other replayers regarding recording time, trace size and replaying time, across multiple target application properties: (i) number of threads, (ii) number of shared memory accesses per thread, (iii) load to store ratio, (iv) number of fields per shared object, and (v) number of shared objects, (vi) number of processors.

The results are presented in Figures 4 and 5. Note that graphs related to execution times use a logarithmic scale due to the order of magnitude-sized differences between replayers' performance, and that in all graphs lower is better.

Figure 4 shows the performance results of application properties (i) to (iii). Record and replay execution times grows linearly with the number of threads, with Ditto taking the lead in absolute values by one and two orders of magnitude, respectively. As for trace file sizes, Ditto stays below 200Mb, while no other replayer comes under 500Mb. The maximum is achieved by LEAP at around 1.5Gb. Concerning the number of memory access operations, the three indicators increase linearly with the number of memory accesses for all algorithms. We attribute this result to two factors: (i) none of them keeps state whose complexity

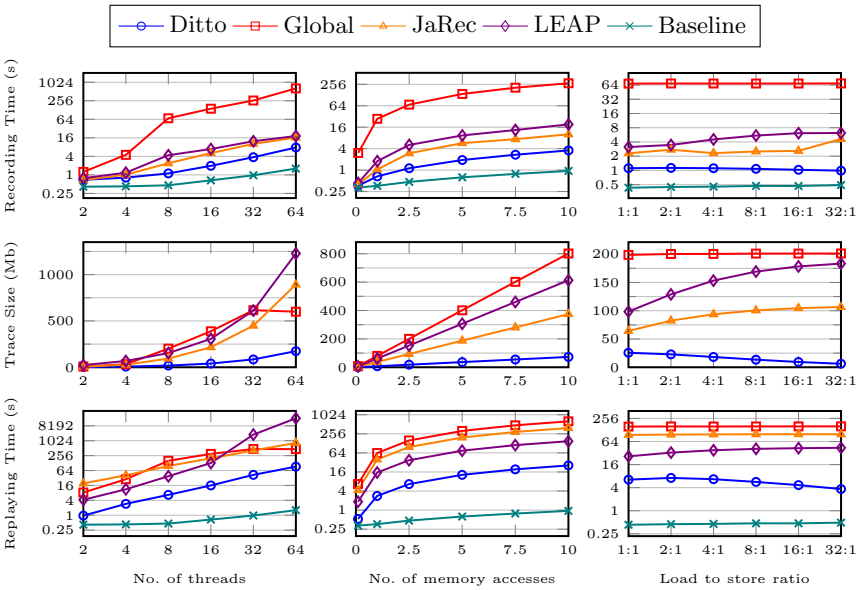


Fig. 4. Recording time, trace size and replaying time as a function of the number of threads, accesses per thread (x10^6) and load:store ratio

increases over time, and (ii) our conscious effort during implementation to keep memory usage constant. Ditto is nonetheless superior in terms of absolute values. Finally, regarding the load and store ratio, Ditto is the only evaluated replayer that takes advantage of the semantic differences between load and store memory accesses. As such, we expect it to be the only system to positively react in the presence of a higher load:store ratio. The experimental results are consistent with this, as we can observe reductions in both overheads and a very significant reduction of the trace file size.

Figure 5 shows the performance results of application properties (iv) to (vi). Stressing the system with an increasing number of fields per object, property (iv), and number of shared objects, property (v), is crucial to measure the impact of Ditto's recording granularity. Ditto and LEAP are the only replayers that improve performance (smaller recording and replaying times) as more shared fields are present, though Ditto has the lowest absolute values. This result is due to both replayers distinguishing between different fields when serializing events. However, LEAP actually increases its trace file size as the number of fields increases, a result we believe to be caused by their access vector-based approach to recording.

Regarding the number of shared objects, JaRec is the main competitor of Ditto as they are the only ones that can distinguish between distinct objects. LEAP's offline transformation approach does not allow it to take advantage from this runtime information. Although JaRec is marginally better than Ditto past the 64 object mark, it fails to take advantage of the number of shared objects during the replay phase.

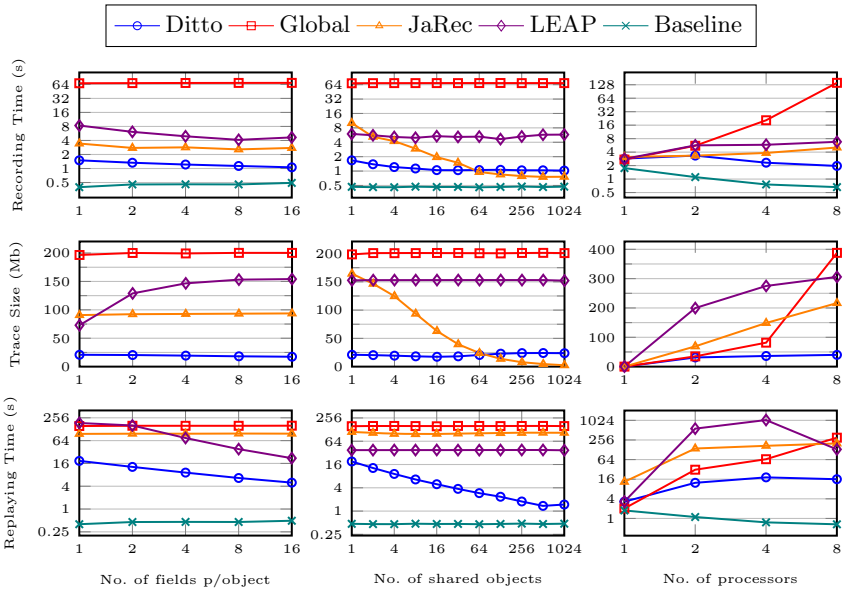


Fig. 5. Recording time, trace size and replaying time as a function of the number of fields per object, shared objects and processors

Concerning the number of processors, the experimental results were obtained by limiting the JikesRVM process to a subset of processors in our 8-core test machine. Ditto is the only algorithm that lowers its record execution time as the number of processors increases, promising increased scalability to future deployments and applications in production environments. Additionally, its trace file size increases much slower than that of other replayers and the replay execution time is three orders of magnitude lower than the second best replayer at the 8 processor mark.

Effects of the Pruning Algorithm: To assess the effects of Ditto's pruning algorithm we modified the microbenchmark to use a more sequential memory access pattern in which each thread accesses a subset of shared objects that overlaps with that of two other threads. Figure 6 shows the trace file size reduction percentage, the recording speedup and the replaying speedup over the base recording algorithm from applying program order pruning only, and program order pruning plus partial transitive reduction. The results clearly demonstrate the potential of the algorithm, reducing the trace by 81.6 to 99.8%. With reductions of this magnitude, instead of seeing increased execution times, we actually observe significant drops in overhead due to the avoided tracing efforts.

Looking at the results of all microbenchmark experiments, it is clear that Ditto is the most well-rounded deterministic replayer. It consistently performs better than its competitors in all three indicators, while other replayers tend to overly sacrifice trace file size or the replay execution time in favor of recording efficiency.

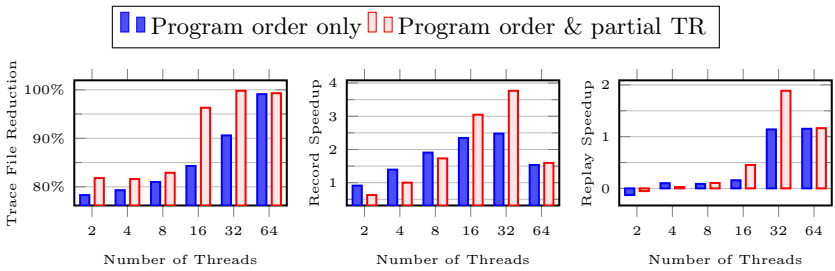


Fig. 6. Effects of Ditto's pruning algorithm

6.2 Complete Applications

In this section we use complete applications to compare the execution time overhead and the log size of Ditto when compared to other state of the art replayers. Furthermore, the impact of the TLO analysis is also evaluated. All applications were parametrized to use 8 threads (i.e. the number of cores of the available hardware). From the the *Java Grande benchmark*[3] we selected the multi-threaded applications, namely: (a) MolDyn, a molecular dynamics simulation; (b) MonteCarlo, a monte carlo simulation; and (c) RayTracer, a 3D ray tracer. Table 1 reports on the results in terms of recording overhead and trace file size. Considering them, two main remarks can be made: Ditto's record-time performance is superior to that of competing replayers, and the trace files generated by Ditto are insignificantly small. The result suggests that the static analysis can be further improved to better identify thread-local memory accesses, which represents a relevant future research topic.

From the *DaCapo*[4] benchmark, we evaluate the record-time performance of Ditto and the other replayers using the lusearch, xalan and avrora applications with the large set. The results are shown in Table 1 and highlight an interesting observation: for applications with very coarse-grained sharing, as is the case of lusearch and xalan, Ditto's higher complexity is actually detrimental. The lack of stress allows the other algorithms to perform better in terms of recording overhead, albeit generating larger trace files (with the exception of JaRec). Nonetheless, Ditto's recording overhead is still quite low.

7 Conclusions and Future Work

We presented Ditto, a deterministic replay system for the JVM, capable of correctly replaying executions of imperfectly synchronized applications on multiprocessors. It uses a novel pair of recording and replaying algorithms that combine state-of-the-art and original techniques, including (a) managing differences between load and store memory accesses, (b) serializing events at instance field

[3] `http://www.epcc.ed.ac.uk/research/java-grande`
[4] `http://dacapobench.org`

Table 1. Record-time performance results for representative Java workloads

	Ditto		Global		JaRec		LEAP	
	Overhead	Trace	Overhead	Trace	Overhead	Trace	Overhead	Trace
MolDyn	2831%	239Kb	>181596%*	>2Gb*	3887%	188Mb	>13956%*	>2Gb
MonteCarlo	390%	248Kb	79575%	1273Mb	410%	0.39Kb	10188%	336Mb
RayTracer	4729%	4.72Kb	>164877%*	>2Gb*	5197%	21Mb	>9697%*	>2Gb*
lusearch	4.56%	3Kb	1.89%	288 Kb	2.26%	3Kb	0.69%	564Kb
xalan	5.23%	6kb	4.52%	475Kb	2.71%	0.2Kb	2.73%	485Kb
avrora	378%	22Mb	2771%	565Mb	372%	23Mb	–*	>2Gb*

* Current implementation cannot deal with trace files over 2 GB.

granularity, (c) pruning redundant constraints using program order and partial transitive reduction, (d) taking advantage of TLO static analysis, escape analysis and compiler optimizations, and (e) applying a simple but effective trace file optimization. Ditto was successfully evaluated to ascertain its capability to reproduce different concurrent bug patterns and highly non-deterministic executions. Performance results show Ditto consistently outperforming previous Java replayers across multiple application properties, in terms of overhead and trace size, being the most well-rounded system, multicore scalable and leveraging checkpointing and restore capabilities. Evaluation results suggest that future efforts to improve deterministic replay should be focused on improving static analysis to identify thread-local events.

Acknowledgments. This work was partially supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, projects PTDC/EIA-EIA/113613/2009, PTDC/EIA-EIA/102250/2008, PEst-OE/EEI/LA0021/2013 and the PROTEC program of the Polytechnic Institute of Lisbon (IPL)

References

1. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. SIGOPS Oper. Syst. Rev. 42, 329–339 (2008)
2. Choi, J.D., Srinivasan, H.: Deterministic replay of java multithreaded applications. In: Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, SPDT 1998, pp. 48–59. ACM (1998)
3. Ronsse, M., De Bosschere, K.: Recplay: a fully integrated practical record/replay system. ACM Trans. Comput. Syst. 17, 133–152 (1999)
4. Georges, A., Christiaens, M., Ronsse, M., De Bosschere, K.: Jarec: a portable record/replay environment for multi-threaded java applications. Softw. Pract. Exper. 34, 523–547 (2004)
5. Dunlap, G.W., Lucchetti, D.G., Fetterman, M.A., Chen, P.M.: Execution replay of multiprocessor virtual machines. In: Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2008, pp. 121–130. ACM (2008)
6. Russinovich, M., Cogswell, B.: Replay for concurrent non-deterministic shared-memory applications. In: Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation, PLDI 1996, pp. 258–266. ACM (1996)

7. Geels, D., Altekar, G., Shenker, S., Stoica, I.: Replay debugging for distributed applications. In: Proceedings of the Annual Conference on USENIX 2006 Annual Technical Conference, p. 27. USENIX Association (2006)
8. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21, 558–565 (1978)
9. Xu, M., Bodik, R., Hill, M.D.: A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In: Proceedings of the 30th Annual International Symposium on Computer Architecture, ISCA 2003, pp. 122–135. ACM (2003)
10. Montesinos, P., Ceze, L., Torrellas, J.: Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In: Proceedings of the 35th Annual International Symposium on Computer Architecture, ISCA 2008, pp. 289–300. IEEE Computer Society (2008)
11. Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R., Lee, K.H., Lu, S.: Pres: probabilistic replay with execution sketching on multiprocessors. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. SOSP 2009, pp. 177–192. ACM (2009)
12. Altekar, G., Stoica, I.: Odr: output-deterministic replay for multicore debugging. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP 2009, pp. 193–206. ACM (2009)
13. Huang, J., Liu, P., Zhang, C.: Leap: lightweight deterministic multi-processor replay of concurrent java programs. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering,l FSE 2010, pp. 207–216. ACM (2010)
14. Yang, Z., Yang, M., Xu, L., Chen, H., Zang, B.: Order: object centric deterministic replay for java. In: Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference, USENIXATC 2011, Berkeley, CA, USA. USENIX Association (2011)
15. Bressoud, T.C., Schneider, F.B.: Hypervisor-based fault tolerance. ACM Trans. Comput. Syst. 14, 80–107 (1996)
16. Napper, J., Alvisi, L., Vin, H.M.: A fault-tolerant java virtual machine. In: DSN, pp. 425–434. IEEE Computer Society (2003)
17. Simão, J., Garrochinho, T., Veiga, L.: A checkpointing-enabled and resource-aware java virtual machine for efficient and robust e-science applications in grid environments. Concurrency and Computation: Practice and Experience 24(13), 1421–1442 (2012)
18. Netzer, R.H.B.: Optimal tracing and replay for debugging shared-memory parallel programs. In: Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging, PADD 1993, pp. 1–11. ACM (1993)
19. Xu, M., Hill, M.D., Bodik, R.: A regulated transitive reduction (rtr) for longer memory race recording. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-XII, pp. 49–60. ACM (2006)
20. Alpern, B., Attanasio, C.R., Barton, J.J., Burke, M.G., Cheng, P., Choi, J.D., Cocchi, A., Fink, S.J., Grove, D., Hind, M., Hummel, S.F., Lieber, D., Litvinov, V., Mergen, M.F., Ngo, T., Russell, J.R., Sarkar, V., Serrano, M.J., Shepherd, J.C., Smith, S.E., Sreedhar, V.C., Srinivasan, H., Whaley, J.: The jalapeño virtual machine. IBM Syst. J. 39(1), 211–238 (2000)