Ditto – Deterministic Execution Replay for the Java Virtual Machine on Multi-processors

João M. Silva
Instituto Superior Técnico - UTL / INESC-ID Lisboa
joao.m.silva@ist.utl.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. Moreover, conventional debugging methodologies are not well suited to deal with non-deterministic faults, leading to inefficient debugging efforts in which most resources are consumed in reproduction attempts. Deterministic replay tackles this problem by recording faulty executions and using the traces to generate equivalent ones. Replayers can be efficient on uni-processor machines, but struggle with unreasonable overhead on multi-processors.

We present Ditto, a deterministic replayer for concurrent JVM applications executed on multi-processor machines. By integrating state-of-the-art and novel techniques it manages to consistently out-perform previous deterministic replayers targeted at Java programs, in terms of recording overhead, replaying overhead and trace file size. The main contribution of Ditto is a novel pair of recording and replaying algorithms that (a) leverage the semantic differences between load and store memory accesses, (b) serialize memory accesses at the instance field level, (c) employ partial transitive reduction and program-order pruning on-the-fly, and (d) take advantage of TLO static analysis, escape analysis and JVM compiler optimizations to identify thread-local accesses.

1. Introduction

For most of computer science's short history, developers have enjoyed significant performance improvements with each new generation of processors, which translated directly to increased software efficiency. As a roadblock on processor speed was reached in the past decade, CPU manufacturers transitioned to improving performance through processor replication, leading to the proliferation of multi-core and multi-processor machines. The performance derived from replication does not, however, immediately increase software efficiency. Indeed, developers must take the effort to identify opportunities for parallelism and rebuild their programs to take advantage of them. This task has been facilitated by the development of concurrent programming models that have become as ubiquitous as the multi-processor machines themselves.

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 avoiding 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 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 [8]. Furthermore, any trace statements added to the program in an effort to learn more about the problem can contribute to the fault's evasiveness. Hence, cyclic debugging becomes even less efficient in the best case, and ineffective in the worst.

Most solutions that tackle the problem of debugging concurrent programs are based upon the idea of eliminating non-deterministic behavior to re-enable conventional debugging methodologies as efficient and effective tools. Deterministic replay has long been suggested as one such solution, based on the idea of recording non-deterministic events during a faulty execution and using the resulting trace to force other executions of the same program to experience equal outcomes of the same events, hence reproducing the fault. A fitting metaphor is that of a time machine, allowing debuggers to inspect past states of a particular execution [6].

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 uni-processors, 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, as in the case of DejaVu [2], RecPlay [12] and JaRec [4].

1.1 The challenge of multi-processors

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 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.

1.2 Ditto, a deterministic replayer for the JVM

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 of Ditto are:

- A novel pair of logical clock-based [7] recording and replaying algorithms that:
  - Leverage the semantic differences between load and store memory accesses to reduce trace data and maximize replay-time concurrency;
  - Serialize memory accesses at the finest possible granularity, distinguishing between distinct static, instance and array fields;
  - Employ a modified version of Netzer's transitive reduction [10] to reduce the amount of trace data on-the-fly;
  - Takes advantage of thread-local objects static analysis, escape analysis and JVM compiler optimizations to reduce the set of monitored memory accesses;
- A trace file optimization that highly reduce the size of logical clock-based traces.

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.

1.3 Document Roadmap

The rest of the paper is organized as follows: Section 2 describes some instances of related work; Section 3 explains the design and algorithms of Ditto; Section 4 presents and discusses evaluation results; and Section 5 concludes the paper and presents our thoughts on the direction 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 [12] 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.

Some researchers have developed specialized hardware-based solutions. FDR [14] extends the cache coherence protocol to propagate causality information and generate an ordering over memory accesses. DeLorean [9] 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 very 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 [1]. 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 recent Java deterministic replayer that employs static analysis to identify memory accesses performed on actual thread-shared variables, hence reducing the amount of monitored accesses [5]. This is a promising technique, but LEAP's recording algorithm associates access vectors to fields, which has significant drawbacks.

Only the first and fourth of these approaches have been employed in the context of Java programs.

3. Ditto

3.1 Events of interest

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 a critical section. 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.

3.2 Base record and replay algorithms

The recording and replaying algorithms of Ditto rely on logical clocks (or Lamport clocks) [7], 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 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.

3.2.1 Recording

The recorder creates two streams of order constraints per thread – one orders shared memory accesses, while the other orders monitor acquisitions.

Recording shared memory accesses 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 written; and (b) a load counter, incremented when

Algorithm 1 Recording memory accesses

```
Parameters: $f$ is the field, $v$ is the value loaded or stored
  method WRAPLOAD($f$,$v$)
      MONITORENTER($f$)
      $t \leftarrow$ GETCURRENTTHREAD()
      TRACE($f.storeClock$)
      $f.loadCount \leftarrow f.loadCount + 1$
      if $f.storeClock > t.clock$ then
          $t.clock \leftarrow f.storeClock$
      end if
      $v \leftarrow$ LOAD($f$)
      MONITOREXIT($f$)
  end method
  method WRAPSTORE($f$,$v$)
      MONITORENTER($f$)
      $t \leftarrow$ GETCURRENTTHREAD()
      TRACE($f.storeClock, f.loadCount$)
      $newClock \leftarrow$ MAX($t.clock, f.storeClock$) $+ 1$
      $f.storeClock \leftarrow newClock$
      $f.loadCount \leftarrow 0$
      $t.clock \leftarrow newClock$
      STORE($f, v$)
      MONITOREXIT($f$)
  end method
```

the field's value is loaded and reset if it is modified. The manipulation of this state and the load or 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. Algorithm 1 lists pseudo-code for these recording processes.

Recording synchronization Unlike memory accesses, which are 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 a second clock to threads. When a thread T_i acquires the monitor of an object o, it

Algorithm 2 Recording monitor acquisition operations

```
Parameters: $o$ is the object whose monitor was acquired
  method AFTERMONITORENTER($o$)
    $t \leftarrow$ GETCURRENTTHREAD()
    TRACE($o.syncClock$)
    $newClock \leftarrow$ MAX($t.syncClock, o.syncClock$) $+ 1$
    $o.syncClock \leftarrow newClock$
    $t.syncClock \leftarrow newClock$
  end method
```

performs Algorithm 2. 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.

3.2.2 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 new identifier to the child thread, its replay identifier. The monitor acquisitions involved are replayed using the same algorithms that handle application-level synchronization, ensuring that replay identifiers remain consistent across executions.

3.2.3 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 (and input) 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. This is 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. 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

Algorithm 3 Replaying load memory access operations

```
Parameters: $f$ is the field whose value is being loaded into $v$
  method WRAPLOAD($f$,$v$)
    $t \leftarrow$ GETCURRENTTHREAD()
    $targetStoreClock \leftarrow$ GETNEXTLOADCONSTRAINT($t$)
    while $f.storeClock < targetStoreClock$ do
      WAIT($f$)
    end while
    $v \leftarrow$ LOAD($f$)
    $t \leftarrow$ GETCURRENTTHREAD()
    if $f.storeClock > t.clock$ then
      $t.clock \leftarrow f.storeClock$
    end if
    $f.loadCount \leftarrow f.loadCount + 1$
    NOTIFYALL($f$)
  end method
```

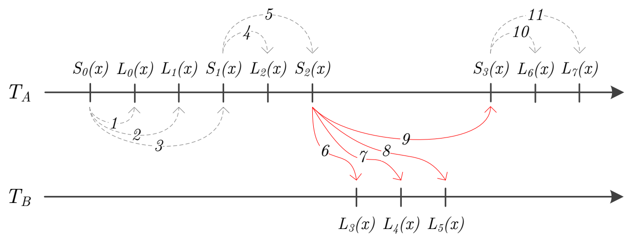
positively re-evaluating the conditions for advancement, it is free to perform the actual load operation. After doing so, thread and field states are updated and waiting threads are notified of the changes. Algorithm 3 lists pseudo-code for this process. If T_i was performing a store operation, the process 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 equalled the respective targets. State would be updated according to the rules used in Algorithm 1.

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 how Ditto allows load operations that read the same value to happen concurrently.

Replaying synchronization 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 2.

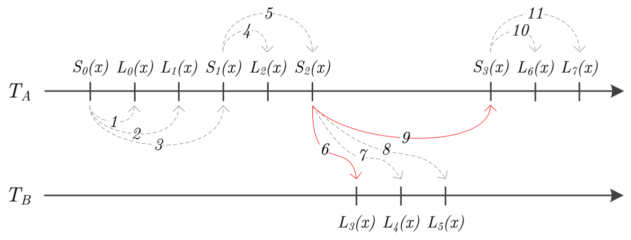
3.3 Recording granularity

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 [5]; 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. However, when this becomes a prob-



5
4
11
10
$S_0(x)$ $L_0(x)$ $L_1(x)$ $S_1(x)$ $L_2(x)$ $S_2(x)$
$S_3(x)$ $L_6(x)$ $L_7(x)$
T_A
1
2
3
9
6
7
8
T_B
$L_3(x)$ $L_4(x)$ $L_5(x)$

(a) Pruning constraints implied by program order.



5
4
11
10
$S_0(x)$ $L_0(x)$ $L_1(x)$ $S_1(x)$ $L_2(x)$ $S_2(x)$
$S_3(x)$ $L_6(x)$ $L_7(x)$
T_A
1
2
3
9
6
7
8
T_B
$L_3(x)$ $L_4(x)$ $L_5(x)$

(b) Pruning constraints implied by previous constraints.

Figure 1. Example usage of Ditto's constraint pruning algorithm.

lem, Ditto is capable of operating with an object-level granularity.

Array fields are treated like object fields, but with a slight twist. To keep field state under control for large arrays, a user-defined cap is placed on how many field states an array can have. Hence, multiple array fields may map to a single field 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.

3.4 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.

3.4.1 Program order pruning

Consider the recorded execution in Figure 1, 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 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. Synchronization order constraints are handled in the same way as load operations, but state is associated with an object instead of a field.

3.4.2 Partial transitive reduction

Netzer introduced an algorithm to find the optimal set of constraints to reproduce an execution [10]. Ditto does not directly employ the algorithm, for reasons related to performance degradation and the need for keeping flexibility-limiting state, such as the usage of vector clocks, requiring the number of threads to be known a priori. We do, however, use it as inspiration for 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 $S_2(x) \to L_3(x)$ 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 1(b).

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 2 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 $S_0(x) \to L_0(x)$. 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

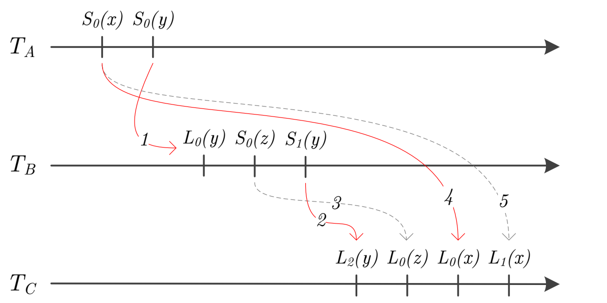


Figure 2. Example of partial transitive reduction.

given that the benefit is limited to one order constraint, as the subsequent constraint 5, similar to 4, is pruned. In essence, Ditto is only aware of thread interactions that span a maximum of one traced order constraint.

3.5 Thread Local Objects Static Analysis

The JVM's memory model provides some guarantees with regards to the locality of variables, namely that method-local variables cannot be shared. Ditto can easily avoid tracing accesses to these variables, as they are accessed by a specific family of bytecodes. There remain, however, a lot of accesses to static and instance fields which are not involved in inter-thread interactions, but about which there is no locality information, forcing us to conservatively consider them a source of non-determinism and trace their outcome.

Thread Local Objects (TLO) static analysis provides locality information on class fields. Its usage in deterministic replay was pioneered by the authors of LEAP [5]. The output of the analysis is a classification of either thread-local or thread-shared for each class field. A stand-alone application 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.

3.6 Array escape analysis

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 field accesses. Ditto uses compile-time escape analysis on array references to avoid monitoring accesses to fields of arrays declared in a method whose reference never escapes that same method. The analysis is very simplistic, but it 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.

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

3.7 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 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 [5], do not benefit from this – the lifetime of a field is the lifetime of the application itself. A thread organized by instance would be even more problematic, as intercepting object creation and collection is not an easy task.

3.7.1 Trace file format

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 stream) 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.

3.7.2 Logical clock value optimization

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.

4. 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 JikesRVM using the same facilities used by Ditto itself, namely (a) DejaVu [2], a global-order replayer; (b) JaRec [4], a partial-order, logical clock-based replayer; and (c) LEAP [5], 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 solely synchronization races, were extended to deal with all data races.

Moreover, all replayers are allowed to benefit from the same static and compile-time analysis used by Ditto to reduce the amount of intercepted events.

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 through comparison between Ditto's runtime performance characteristics and those of the other implemented replayers. The results are gathered by performing a microbenchmark and recording execution of selected applications 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.

4.1 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 [1].

Microbenchmark The microbenchmark was designed 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 is available online[3].

IBM concurrency testing repository The repository contains a number of small applications that exhibit various concurrent bug patterns while performing some practical task. Table 1 lists the evaluated applications along with the concurrent bug patterns they exhibit [3]. Ditto is capable of correctly reproducing each and every one of these bugs. Although these applications do not constitute the whole benchmark suite, they do amount to the subset of it which does not rely on input non-determinism, as input reproduction is outside Ditto's scope and a solved research problem [13].

4.2 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 the 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 and LEAP.

Application	**Bug Pattern**
Account	Wrong Lock or No Lock
Airline Tickets	Non-atomic Operation
Booking	Non-atomic Operation
Bounded Buffer	`notify` instead of `notifyAll`, Deadlock
Bubble Sort	Non-atomic Operation, Orphaned Thread
Linked List	Non-atomic Operation
Liveness	Dormancy, Lost `notify`
Loader	`sleep` Interleaving
Lottery	Non-atomic Operation, Wrong Lock or No Lock
Manager	Non-atomic Operation
Merge Sort	Non-atomic Operation
Pingpong	Wrong Lock or No Lock
Piper	Condition for `wait`, Deadlock
Prod. Consumer	Orphaned Thread
Shop	`sleep` Interleaving, Double-checked Locking

Table 1. Summary of evaluated applications from the IBM Concurrency Testing Repository

4.2.1 Microbechmark

The same microbenchmark used to assess replay correctness is now used to compare Ditto's performance characteristics with those of the other replayers, 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. The results are presented in Figures 3 and 4. Note that graphs related to execution times use a logarithmic scale due to the order of magnitude-sized differences between replayers' performance.

Effect of the number of threads Record and replay execution times grow 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, with the maximum being achieved by LEAP at around 1.5Gb.

Effect of 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 increases over time, and (ii) our conscious effort dur-

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

[3] `https://github.com/gdeOo/ditto`

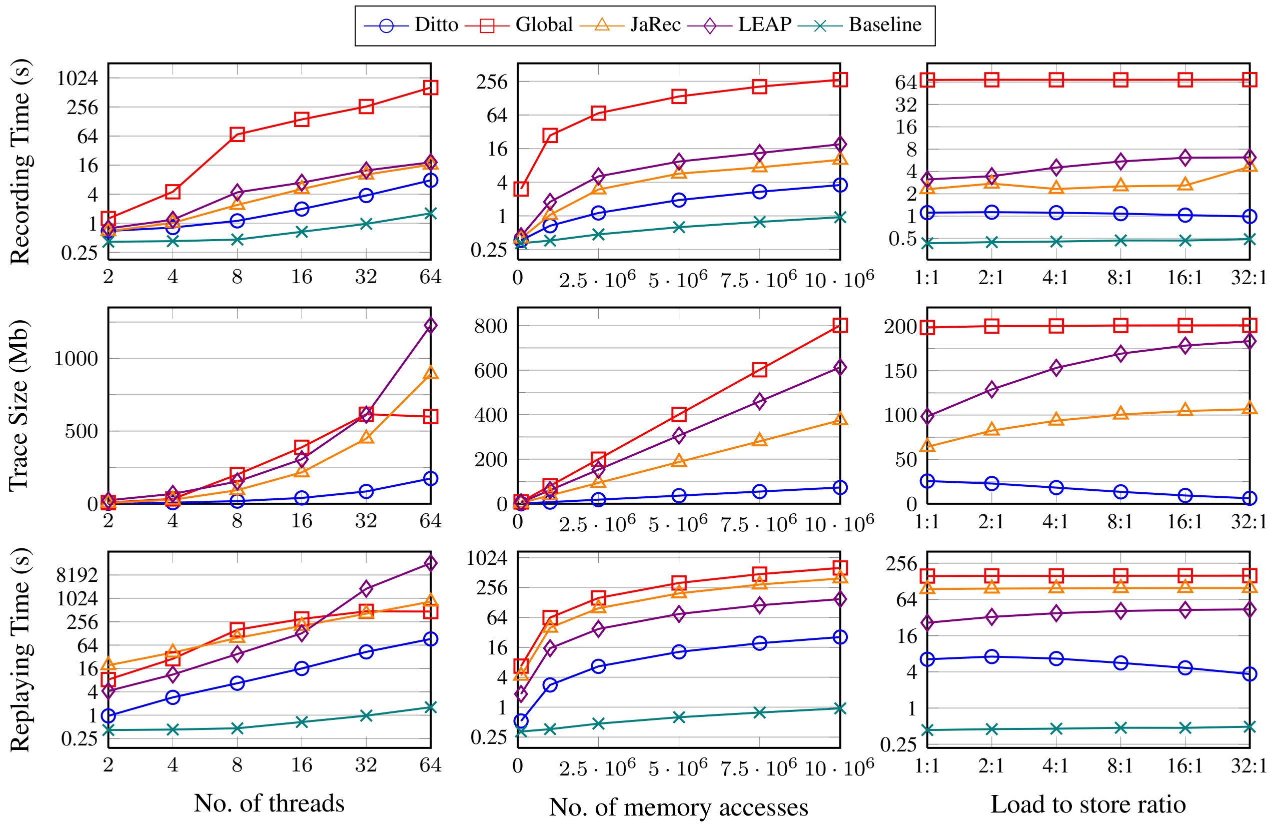


Figure 3. Microbenchmark's results as a function of the number of threads, accesses per thread and load:store ratio.

ing implementation to keep memory usage constant. Ditto is nonetheless superior in terms of absolute values.

Effect of the load to 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 replayer to positively react in the presence of a higher load:store ratio. The experimental results are consistent with this expectation, as we can observe reductions in both overheads and a very significant reduction of the trace file size.

Effect of the number of fields per shared object Given that only Ditto and LEAP can distinguish between different fields, we expect them to be the only replayers that improve performance as more shared fields are present, reducing contention. This is indeed what happens in terms of recording and replaying execution times. 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.

Effect of the number of shared objects Our expectations are similar to those of the previous experiment, but with JaRec in place of LEAP, as it is the only replayer besides Ditto that distinguishes between distinct instances. This is the only axis along which Ditto is overtaken by a competitor, with JaRec taking the lead in recording execution time and trace file size past the 64 object mark. However, JaRec fails to take advantage of the number of shared objects during replay.

Effect of 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.

Trace file compression Trace files generated by Ditto can greatly benefit from compression algorithms. Compressing the trace files generated during the microbenchmark experiments with gzip[4] yielded an average compression rate of 41.8% with a standard deviation of 7.0%.

Overall observations 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

[4] `http://www.gzip.org`

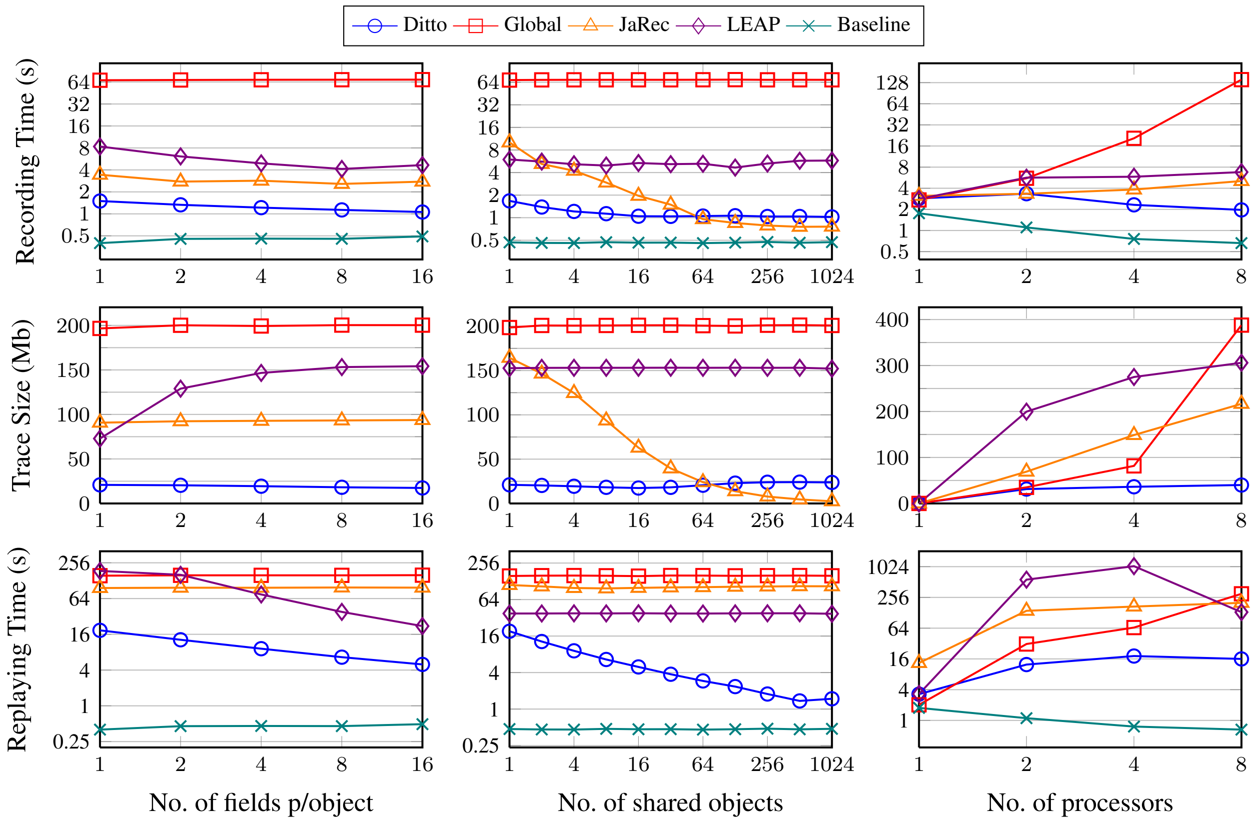


Figure 4. Microbenchmark's results as a function of the number of fields per object, shared objects and processors.

replayers tend to overly sacrifice trace file size or the replay execution time in favor of recording efficiency.

4.2.2 Java Grande benchmark

The Java Grande benchmark suite[5] contains "Grande" applications, with large requirements in either memory, bandwidth or processing power. The multi-threaded version of the suite contains three applications: (a) MolDyn, a molecular dynamics simulation; (b) MonteCarlo, a monte carlo simulation; and (c) RayTracer, a 3D ray tracer. Table 2 reports on the results in terms of recording overhead and trace file size. Considering them, two main remarks can be made. To start with, even though Ditto's record-time performance is superior to that of competing replayers, its recorder is still unreasonably inefficient for production environments. The second is that the trace files generated by Ditto are insignificantly small. Considering the high recording overhead, this can only mean that most monitored memory accesses were not involved in inter-thread interactions, a fact which is not visible in the trace files of other replayers, with the exception of JaRec's recording of MonteCarlo. The result suggests that the static analysis performed on the application did not do a satisfying job in identifying thread-local memory accesses.

[5] `http://www.epcc.ed.ac.uk/research/java-grande`

4.2.3 DaCapo benchmark

The DaCapo benchmark suite[6] consists of a set of real world applications, many of which are concurrent, exhibiting different levels of inter-thread interaction granularity. We evaluate the record-time performance of Ditto and the other replayers using the lusearch, xalan and avrora applications from the 9.12-bach version of DaCapo, with their default parameters. The results are shown in Table 2 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.

5. Conclusions and Future Work

We presented Ditto, a deterministic replay system for the JVM, capable of correctly replaying executions of imperfectly synchronized applications on multi-processors. It uses a novel pair 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 granular-

[6] `http://dacapobench.org`

	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.

Table 2. Record-time performance results for the MolDyn benchmark of the Java Grande suite.

ity, (c) pruning redundant constraints using program order and partial transitive reduction, (d) taking advantage of TLO static analyses, 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 that Ditto consistently outperforms previous Java replayers across multiple application properties, in terms of overhead and trace size, being the most well-rounded system. Nonetheless, Ditto's recording overhead is still too high for production environments when targeting applications with fine-grained inter-thread interactions. 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, under projects PTDC/EIA-EIA/113613/2009, PTDC/EIA-EIA/102250/2008 and PEst-OE/EEI/LA0021/2011.

References

[1] Gautam Altekar and Ion Stoica. Odr: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 193–206. ACM, 2009.

[2] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, SPDT '98, pages 48–59. ACM, 1998.

[3] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. IEEE Computer Society, 2003.

[4] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. Jarec: a portable record/replay environment for multi-threaded java applications. *Softw. Pract. Exper.*, 34:523–547, May 2004.

[5] Jeff Huang, Peng Liu, and Charles Zhang. Leap: lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 207–216. ACM, 2010.

[6] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 1–1. USENIX Association, 2005.

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

[8] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGOPS Oper. Syst. Rev.*, 42: 329–339, March 2008.

[9] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 289–300. IEEE Computer Society, 2008.

[10] Robert H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, PADD '93, pages 1–11. ACM, 1993.

[11] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. Pres: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 177–192. ACM, 2009.

[12] Michiel Ronsse and Koen De Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17:133–152, May 1999.

[13] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 3–3. USENIX Association, 2004.

[14] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 122–135. ACM, 2003.