Efficient Deterministic Replay of Multithreaded Executions in a Managed Language Virtual Machine [*]

Michael D. Bond
Ohio State University
mikebond@cse.ohio-state.edu

Milind Kulkarni
Purdue University
milind@purdue.edu

Man Cao
Ohio State University
caoma@cse.ohio-state.edu

Meisam Fathi Salmi
Ohio State University
fathi@cse.ohio-state.edu

Jipeng Huang
Microsoft
jiphuang@microsoft.com

Abstract

Shared-memory parallel programs are inherently nondeterministic, making it difficult to diagnose rare bugs and to achieve deterministic execution. Existing multithreaded record & replay approaches have serious limitations such as relying on custom hardware, handling only data-race-free executions, or slowing programs by an order of magnitude. Furthermore, language virtual machines (VMs) such as Java VMs (JVMs) introduce various sources of nondeterminism that thwart demonstrating deterministic replay.

This paper introduces an approach for multithreaded record & replay based on tracking and reproducing shared-memory dependences accurately and efficiently. Building on prior work that introduces an efficient dependence *recorder*, we develop a new analysis for *replaying* dependences. To demonstrate multithreaded record & replay, we modify a JVM to support a new methodology that enables demonstrating and evaluating replay in the inherently nondeterministic JVM. Overall, the performance of both recorded and replayed executions compares favorably with performance reported by prior work for competing record & replay approaches.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Debuggers, Run-time environments

Keywords record & replay; language virtual machines

1. Introduction

Shared-memory programs are inherently nondeterministic because memory accesses interleave in different ways. Nondeterminism makes it difficult to diagnose production-time errors and to execute replicated multithreaded processes.

Researchers have proposed *record & replay* to address this challenge [2, 20, 22, 24–26, 29–31, 34, 36, 39, 42, 43, 46, 48, 51, 52]. One execution records enough information to allow another execution to replay the same thread interleavings faithfully, thus achieving the same execution result. Record & replay can support *offline* replay, *online* replay, or both. Offline replay supports replaying a recorded execution at a later time, enabling debugging of production-time errors. Online replay executes the replayed run concurrently with the recorded run, enabling replication-based fault tolerance [13] and distribution of dynamic analysis among multiple execution instances [17, 40].

In order to replay multithreaded executions faithfully, record & replay approaches must record how threads interleave, which includes not only the order of synchronization operations (e.g., the order that two threads acquire the same lock), but also the order of unsynchronized memory accesses (i.e., loads and stores involved in *data races*)—which is expensive since many accesses can potentially be unsynchronized. Existing record & replay approaches incur high overhead to track cross-thread dependences [29, 30], cannot handle racy executions [22, 46], rely on speculation and extra cores [31, 48], support online or offline replay but not both [2, 26, 31, 34, 42, 51], or rely on custom hardware [24, 25, 36, 39, 52].

This paper builds on prior work called *Octet*, which introduces efficient monitoring of cross-thread dependences [11]. However, that work does not describe nor provide support for *replaying* dependences. Replaying dependences is challenging for two reasons. First, it is not straightforward *how* to record and replay dependences captured by Octet. Second, *demonstrating* and *evaluating* REPLAY is challenging, particularly in a managed language virtual machine (VM) such as a Java VM (JVM) that has many sources of nondeterminism that affect application determinism.

Contributions. This paper makes two main contributions. First, we design, implement, and evaluate an analysis for replaying dependences recorded by prior work's dependence recorder [11]. Second, in order to demonstrate and evaluate replay, we modify a JVM to support a new methodology that eliminates most of the JVM's nondeterminism. These contributions enable us to demonstrate a new record & replay approach that efficiently and flexibly handles racy programs, targets commodity systems, supports both online and offline replay, and adds low overhead—properties that existing techniques have struggled to provide simultaneously.

Our record & replay approach introduces two dynamic analyses called RECORD and REPLAY. RECORD identifies and records dependences by using Octet to track dependences [11]. Our new REPLAY analysis executes the program in parallel, enforcing the

[*] This material is based upon work supported by the National Science Foundation under Grants CSR-1218695, CAREER-1253703, and CCF-1421612.

PPPJ '15, September 8–11, 2015, Melbourne, FL, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3712-0/15/09...$15.00.
http://dx.doi.org/10.1145/2807426.2807434

cross-thread data dependences recorded by RECORD. REPLAY does *not* use Octet and is a distinctly different analysis from RECORD. REPLAY necessarily *elides* program synchronization operations, which would otherwise conflict nondeterministically with the recorded cross-thread data dependences.

We have implemented RECORD and REPLAY in Jikes RVM [1], a JVM that targets research but performs competitively with commercial JVMs [7]. In order to demonstrate REPLAY, we have (i) implemented novel approaches for controlling sources of nondeterminism in the JVM and (ii) introduced a research methodology that helps further limit nondeterminism. This support for determinism, which enables us to demonstrate and evaluate REPLAY, is not immediately suitable for production use; however, this limitation is not inherent to our record & replay approach. Furthermore, while the implementation currently supports only offline replay, the approach is suited to providing both offline and online replay.

Because REPLAY elides synchronization and enforces the same dependences as RECORD, REPLAY is often faster than RECORD. Overall, RECORD and REPLAY add lower overhead than reported by prior work that provides the same features (i.e., supports online and offline replay in commodity systems).

Section 2 motivates the challenges of multithreaded record & replay. Section 3 overviews our RECORD and REPLAY analyses. Section 4 overviews the RECORD analysis that is largely based on prior work. Section 5 introduces our new REPLAY analysis; Section 5.3 argues that REPLAY soundly replays dependences recorded by RECORD. Section 6 describes JVM modifications to deal with nondeterminism and a methodology for demonstrating multithreaded replay. Section 7 evaluates the characteristics, effectiveness, and performance of our record & replay system. Section 8 covers prior work related to our contributions.

2. Background and Motivation

This section motivates the challenge of recording and replaying nondeterministic multithreaded behavior. From a performance perspective, record & replay is straightforward for *single-threaded* executions (or multithreaded executions on a single core): sources of nondeterminism—such as I/O, system time, thread context switches, and other system-level effects—are infrequent enough that recording and replaying them adds low overhead. However, it is considerably more challenging to make record & replay efficient for multithreaded, shared-memory programs because many program operations can be involved in nondeterministic interactions between threads.

Threads interleave nondeterministically at so-called *high-level races* (races on synchronization operations) and *data races* (races on ordinary loads and stores). While synchronization operations tend to be infrequent enough that they can be recorded and replayed efficiently, many loads and stores can potentially be involved in data races, making it expensive to capture all thread interactions.

Data races are common in real software because they are easy to introduce, expensive to detect, and hard to eliminate (Section 8.1). Prior approaches either ignore data races but are thus unsound [22, 46]; sidestep the challenge of catching racy interleavings but incur serious limitations [2, 26, 31, 34, 42, 48, 51]; or track dependences explicitly but incur high overhead or other significant limitations [20, 27, 29, 30, 34, 53]; or rely on custom hardware support [24, 25, 36, 39, 43, 52] (Section 8.1).

To replay potentially racy executions faithfully, it is sufficient to reproduce all cross-thread data dependences: data dependences (write–read, read–write, and write–write dependences) involving two threads. Recording all cross-thread dependences is expensive for several reasons. (1) Every access to potentially shared memory must be instrumented. (2) The instrumentation typically involves tracking the last thread(s) to write and/or read each potentially shared variable, leading to remote cache misses at mostly-read-shared program accesses. (3) To track dependences accurately, the instrumentation itself must use synchronization to ensure atomicity of the instrumentation together with the program access.

Our prior work called *Octet* tracks cross-thread dependences efficiently by giving up on precise tracking of dependences—instead establishing *happens-before edges* that together imply all cross-thread dependences [11]—in exchange for significantly better performance. Section 4 describes the RECORD analysis built on top of Octet. However, a significant remaining challenge, which this paper addresses, is how to *replay* the happens-before edges recorded by RECORD. Part of the challenge is how synchronization operations, which are not tracked by RECORD, should be handled during the replayed execution. The replayed execution must be efficient and scalable—at least matching the performance of the recorded execution—in order to support online replay.

A second challenge this paper addresses is the difficulty of demonstrating and evaluating REPLAY, particularly in a language virtual machine (VM) such as a Java VM (JVM), which introduces many sources of nondeterminism that confound replaying an execution deterministically. Section 8.3 describes prior work on demonstrating replay and dealing with nondeterminism in JVMs.

Addressing these challenges yields a record & replay approach that advances the state of the art, by outperforming competing approaches (i.e., approaches that support online and offline replay of racy programs in commodity systems).

3. Overview and Preliminaries

Our record & replay approach consists of two distinct analyses, RECORD and REPLAY, described separately in Sections 4 and 5. RECORD and REPLAY operate on two separate executions; these executions can either execute concurrently (providing online replay), or REPLAY can execute after RECORD (providing offline replay).

RECORD identifies and records *happens-before* edges that each involve a *source* point on one thread and a *sink* point on another thread. During RECORD, each program thread T records information in its per-thread log, T.log, to enable replaying the source or sink of a happens-before edge. During REPLAY, each program thread T reads from T.log in order to replay the sources and sinks of happens-before edges.

RECORD and REPLAY need to agree on *when* events occur in program execution. They maintain a *dynamic program location* (DPL) for each thread to represent a dynamic program point uniquely. We represent DPL as (1) a static site (e.g., method and bytecode index) and (2) a per-thread counter, T.dynCtr, incremented on every loop back edge, method entry, and method return.

4. Recording Cross-Thread Dependences

This section presents a dynamic analysis called RECORD that records *happens-before* edges that soundly imply all cross-thread dependences in an execution. RECORD builds on an existing dynamic analysis from our prior work called Octet [11]. The Octet paper outlines, implements, and evaluates an approach for recording dependences identified by Octet [11], although it does not actually present details of an analysis for recording replayable dependences. The rest of this section overviews Octet and presents the details of the RECORD analysis.

4.1 Tracking Cross-Thread Dependences

Octet is a dynamic analysis that establishes *happens-before* relationships [28] that soundly imply all *cross-thread dependences*, which are data dependences (write–read, read–write, and write–write dependences) involving two threads. Although Octet instruments all accesses, it achieves low overhead by making an opti-

Code path	Transition type	Old state	Access	New state	Record HB edge(s)?
Fast	None	$\mathsf{WrEx_T}$	R or W by T	Same	No
		$\mathsf{RdEx_T}$	R by T	Same	
		$\mathsf{RdSh_c}$	R by T*	Same	
Slow	Conflicting	$\mathsf{WrEx_{T1}}$	W by T2	$\mathsf{WrEx_{T2}}$	Yes
		$\mathsf{WrEx_{T1}}$	R by T2	$\mathsf{RdEx_{T2}}$	
		$\mathsf{RdEx_{T1}}$	W by T2	$\mathsf{WrEx_{T2}}$	
		$\mathsf{RdSh_c}$	W by T	$\mathsf{WrEx_T}$	
	Upgrading	$\mathsf{RdEx_T}$	W by T	$\mathsf{WrEx_T}$	No
		$\mathsf{RdEx_{T1}}$	R by T2	$\mathsf{RdSh_c}$	Yes
	Fence	$\mathsf{RdSh_c}$	R by T*	Same*	Yes

Table 1. Octet's state transitions establish happens-before edges. The last column shows which happens-before edges RECORD records. *A read by T to an object in $\mathsf{RdSh_c}$ state triggers a fence transition if a per-thread counter $\mathsf{T.rdShCount} < \mathsf{c}$. A fence transition updates $\mathsf{T.rdShCount} \leftarrow \mathsf{c}$.

mistic tradeoff: an access *not* involved in cross-thread dependences can use cheap, unsynchronized instrumentation; but an access involved in cross-thread dependence(s) requires expensive coordination among threads.

The Octet analysis identifies cross-thread dependences by tracking the *state* of each potentially shared object,[1] based on the last thread(s) to write or read the object. Before each program memory access, the analysis uses this state to determine if the access might be involved in a cross-thread dependence. An access that does *not* require a state change is definitely *not* involved in a cross-thread dependence. Each object has one of the following states at any time:

$\mathsf{WrEx_T}$: Write exclusive for thread T. T may read or write the object without changing the state. Newly allocated objects start in $\mathsf{WrEx_T}$ state, where T is the allocating thread.

$\mathsf{RdEx_T}$: Read exclusive for thread T. T may read but not write the object without changing the state.

$\mathsf{RdSh_c}$: Read shared. Any thread T may read the object without changing the state, subject to an up-to-date counter $\mathsf{T.rdShCount} \geq \mathsf{c}$ (which helps to ensure that dependences with the last write to the object are captured soundly [11]; Section 4.2).

When a thread attempts to access an object, compiler-inserted instrumentation checks the object's state and updates the state if necessary to allow the access. Table 1 shows the state transition for each possible initial state and access type. Some accesses require no state transition (rows labeled "None"); these accesses are the common case in practice. Other accesses trigger a state change; these accesses may be involved in cross-thread dependences. When a program memory access triggers a state change, it requires either a *conflicting*, *upgrading*, or *fence* transition, described in Section 4.2.

The following pseudocode shows the instrumentation that Octet adds at each program load or store to track per-object states. The analysis metadata o.state represents the state for the object referenced by o. Octet adds the following code at each program store:

```
if (o.state != WrEx_T) {
  /* Slow path: change o.state & call RECORD hooks */
}
o.f = ...; // program store
```

and at each program load:

```
if (o.state != WrEx_T && o.state != RdEx_T &&
    !(o.state == RdSh_c && T.rdShCount >= c)) {
  /* Slow path: change o.state & call RECORD hooks */
}
... = o.f; // program load
```

[1] This paper uses the term "object" to refer to any unit of shared memory.

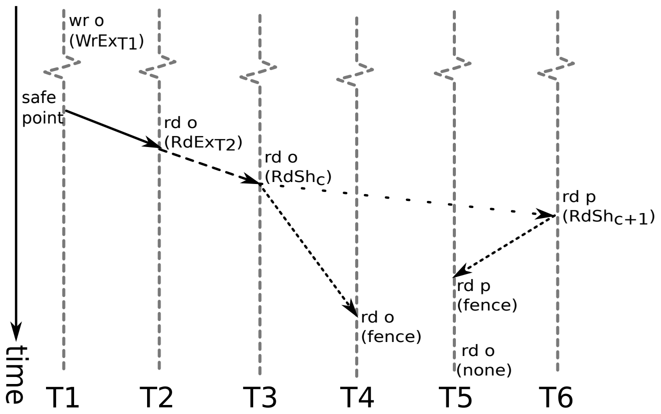

Figure 1. Example execution illustrating state transitions.

As the reader can see, Octet's instrumentation is optimized for accesses that do not trigger a state change—these accesses take the instrumentation "fast path." Other accesses trigger the "slow path," which performs state transitions. Octet establishes happens-before edges for these transitions, which RECORD identifies and records. Next we describe how these transitions work.

4.2 Recording Happens-Before Edges

This section describes how Octet state transitions establish happens-before edges, and how RECORD hooks onto these transitions to identify and record happens-before edges in per-thread logs, using Figure 1 as a running example.

Conflicting transitions. If an access to an object by a thread *conflicts* with the object's state, the access's instrumentation triggers a *conflicting* transition (middle rows of Table 1). In Figure 1, suppose thread T1 has previously written to an object o, so o's state is $\mathsf{WrEx_{T1}}$. Before T2 performs a load from o, RECORD's instrumentation triggers a conflicting transition.

T2 cannot simply change o's state—even if it uses synchronization—since it might race with T1 continuing to perform unsynchronized accesses to o, potentially missing cross-thread dependences. To handle the conflicting transition correctly, T2 *coordinates* with T1 to ensure that T1 does not continue accessing o. (At a write to an object in $\mathsf{RdSh_c}$ state, a thread coordinates separately with every other thread.) Performing coordination establishes a happens-before relationship that implies the dependence from T1's last access of o to T2's current load of o. T1 only *responds* to T2's *request* when T1 is at a *safe point*: a point that is definitely not between an access and its corresponding instrumentation. If T1 is blocked (e.g., waiting for a lock or for a coordination response), then T2 coordinates with T1 "implicitly," ensuring progress [11].

Figure 1 shows the happens-before edge that RECORD must capture: from a safe point on T1 to T2's load. RECORD piggybacks on coordination in order to record the source and sink of this happens-before edge. T1's safe point records the *source* of this happens-before edge by executing the following pseudocode, which records the thread's current DPL:

```
T.log.recordEvent(RESPONSE, currentSiteID, T.dynCtr);
```

where T is the current thread, and currentSiteID identifies the current static program location (method and bytecode index). T.log.recordEvent() records an event identifier (e.g., RESPONSE) and any other arguments in T's file-system-based log.

To record the happens-before *sink* in Figure 1, T2 records its current DPL and the value of a counter T1.responses: the number of coordination responses that T1 has responded to so far (incremented by T1 at each response [11]). T2 executes the following pseudocode:

```
T.log.recordEvent(REQUEST, currentSiteID, T.dynCtr,
      sourceThread.responses);
// Helps with recording upgrading transitions:
if (isRead)
  T.numConflReads++;
```

where sourceThread is the responding thread (T1 in Figure 1). The conditional increment of the per-thread counter T.numConflReads helps with recording upgrading transitions, described next.

Upgrading transitions. An *upgrading* transition expands the set of allowable accesses compared with accesses allowed under the old state. In Figure 1, before T3 performs a read to o in the $\mathsf{RdEx}_{\mathsf{T2}}$ state, it (atomically) upgrades o's state to $\mathsf{RdSh}_{\mathsf{c}}$. The value c that is part of the new $\mathsf{RdSh}_{\mathsf{c}}$ state is the current value of a global counter gRdShCtr that each upgrading transition to RdSh increments atomically. Threads use the counter c to determine whether they have already read an object in $\mathsf{RdSh}_{\mathsf{c}}$ state—or some other object in another state $\mathsf{RdSh}_{\mathsf{c'}}$, where c'>c. This check ensures that happens-before edges are established from the prior write to o to each thread's subsequent read of o.

On an upgrading transition to a RdSh state, such as the transition from $\mathsf{RdEx}_{\mathsf{T2}}$ to $\mathsf{RdSh}_{\mathsf{c}}$ in Figure 1, RECORD must record two happens-before edges:

1. A happens-before edge from the DPL on T2 that changed the same object to $\mathsf{RdEx}_{\mathsf{T2}}$ state. This happens-before edge is needed in order to transitively capture the cross-thread dependence from the last write (by T1 in this case) to T3's read. Identifying this happens-before edge is difficult because the DPL on T2 that changed the object to $\mathsf{RdEx}_{\mathsf{T2}}$ is no longer known when T3's upgrading transition happens. Instead, RECORD records a more conservative happens-before edge: from T2's last transition *of any object* to $\mathsf{RdEx}_{\mathsf{T2}}$.
2. A happens-before edge from the previous upgrade of any object to RdSh, i.e., from the upgrade to $\mathsf{RdSh}_{\mathsf{c}-1}$ (not shown in the figure). This happens-before edge is needed to transitively capture all write–read dependences captured via fence transitions (described shortly). For example, it is necessary to capture the happens-before edge between transitions to $\mathsf{RdSh}_{\mathsf{c}}$ and $\mathsf{RdSh}_{\mathsf{c}+1}$ in order to transitively capture the dependence from T1's write of o to T5's read of o. RECORD records this edge by recording the new value of gRdShCtr, i.e., c.

The current thread T records both of these happens-before edges using the following pseudocode:

```
T.log.recordEvent(UPGRADING, currentSiteID, T.dynCtr,
      sourceThread.numConflReads, c);
```

where c is the result of atomically incrementing gRdShCtr, and sourceThread is the thread such that the object's old state is $\mathsf{RdEx}_{\mathsf{sourceThread}}$.

A *same-thread* upgrading transition from $\mathsf{RdEx}_{\mathsf{T}}$ to $\mathsf{WrEx}_{\mathsf{T}}$ does *not* require recording happens-before edges (Table 1). Any cross-thread dependences are implied by happens-before relationships established as part of the prior transition to $\mathsf{RdEx}_{\mathsf{T}}$ [11].

Fence transitions. Next in Figure 1, T4 reads o, triggering a *fence* transition because T4's thread-local read-shared counter T4.rdShCount $<$ c. The fence transition establishes a happens-before relationship with T3's transition to $\mathsf{RdSh}_{\mathsf{c}}$ and updates T.rdShCount to c. When T5 reads o, T5 has already read an object p in state $\mathsf{RdSh}_{\mathsf{c}+1}$, so *no* fence transition is triggered. However, a transitive happens-before relationship with the prior write to o has been established transitively by (1) the happens-before edge from o's $\mathsf{RdSh}_{\mathsf{c}}$ transition to p's $\mathsf{RdSh}_{\mathsf{c}+1}$ transition, (2) the fence transition on T6 when accessing p, and (3) program order on T5.

At a fence transitions for an object in $\mathsf{RdSh}_{\mathsf{c}}$ state, RECORD records the happens-before edge from (1) the last upgrade to $\mathsf{RdSh}_{\mathsf{c}}$ to (2) the current program point. Figure 1 shows two fence transitions: one establishes a happens-before edge from T3 to T4, and the other establishes an edge from T6 to T5. A thread records a fence transition by recording the current DPL and the value of c in the $\mathsf{RdSh}_{\mathsf{c}}$ state:

```
T.log.recordEvent(FENCE, currentSiteID, T.dynCtr, c);
```

where c is the value in the object's state $\mathsf{RdSh}_{\mathsf{c}}$.

5. Replaying Cross-Thread Dependences

This section describes REPLAY, a new dynamic analysis that enforces happens-before edges recorded by RECORD.

REPLAY does *not* track Octet states. It maintains DPL and reads from per-thread logs (Section 3). At program memory accesses and safe points, REPLAY perturbs execution to replay the same sources and sinks of happens-before edges as during RECORD.

Replaying happens-before edges between memory accesses is sufficient to enforce the same thread interleavings as during a recorded run, so the replayed execution need *not* perform program synchronization operations. In fact, the replayed execution *must* elide synchronization operations in order to avoid deadlock, as Section 5.2 explains.

5.1 Replaying Recorded Happens-Before Edges

REPLAY replays the same happens-before edges that RECORD recorded. It must replay these happens-before relationships both soundly and precisely. Missing a relation could lead to different interleavings; strengthening a relation could lead to deadlock.[2] During REPLAY, each thread reads from the same per-thread log as during RECORD. At a high level, a thread replays the source of a happens-before edge by incrementing some counter (depending on the type of edge recorded) at the same DPL as during RECORD. A thread replays the sink of a happens-before edge by waiting, at the same DPL as during RECORD, for the appropriate counter of the source thread to reach the recorded value.

[2] Note that RECORD is imprecise: it may record spurious happens-before edges that do not represent real dependences in the program. Nevertheless, these edges represent a valid execution of the program, so as long as REPLAY faithfully reproduces them, it will produce the correct result. "Strengthening" in this context means introducing additional ordering constraints beyond what RECORD identifies.

Fast-path instrumentation. For accesses that do not trigger a transition, RECORD records nothing. REPLAY optimizes for this case by performing the following "fast-path" instrumentation at each program memory access and safe point:

```
if (T.log.nextDynCtr == T.dynCtr &&
    T.log.nextSiteID == currentSiteID) {
  // Slow path: replay happens-before edge(s)
  switch (T.log.nextEventType) {
    ...
  }
}
... = o.f;  // program memory access
```

This check succeeds only if the current DPL matches the DPL of the next recorded event. If so, the instrumentation executes the "slow path," which replays happens-before edge(s) from one of the following cases.

Conflicting transitions. Replaying a conflicting transition involves replaying both the source and sink of the established happens-before edge. Referring back to Figure 1 as an example, REPLAY replays the happens-before edge from T1's safe point to T2's load. REPLAY uses the following slow-path instrumentation to replay the source of the happens-before edge:

```
case RESPONSE:
  memory_fence;
  T.responses++;
  T.log.readNextEntry();
  break;
```

The memory fence helps ensure visibility from the source to the sink of the happens-before edge. The readNextEntry() operation reads the next event from the log and updates the variables T.log.nextDynCtr, T.log.nextSiteID, T.log.nextEventType, and other event data (depending on the event type).

To replay the sink of the happens-before edge, slow-path instrumentation performs the following instrumentation, which waits for the source thread's responses counter to "catch up" with the recorded value:

```
case REQUEST:
  S = T.log.nextSourceThread;
  while (S.responses < T.log.nextExpectedResponses) {
    memory_fence;
  }
  if (isRead)              // Assist replay of
    T.numConflReads++;  // upgrading transitions
  memory_fence;
  T.log.readNextEntry();
  break;
```

where T.log.nextSourceThread is the recorded responding thread, and T.log.nextExpectedResponses is the recorded value of S.responses. As part of replaying a transition to RdEx_{T}, T increments the counter T.numConflReads, just as RECORD does, in order to help replay upgrading transitions.

We note that in the above and following instrumentation, whenever REPLAY waits busily in a while loop, it minimizes the cost of busy waiting by switching to non-busy waiting, e.g., waiting on a monitor, after a threshold amount of time has elapsed (not shown in the pseudocode).

Upgrading transitions. To replay an upgrading transition, a thread must replay two recorded happens-before edges: one from the last reader thread, and one that globally orders gRdShCtr increments. The following pseudocode shows how the instrumentation slow path replays these happens-before edges:

```
case UPGRADING:
  S = T.log.nextSourceThread;
  while (S.numConflReads < T.log.nextNumConflReads) {
    memory_fence;
  }
  while (gRdShCtr < T.log.nextRdShCtr − 1) {
    memory_fence;
  }
  gRdShCtr = T.log.nextRdShCtr;
  memory_fence;
  T.log.readNextEntry();
  break;
```

where T.log.nextNumConflReads is the recorded value of S.numConflReads, and T.log.nextRdShCtr is the recorded value of gRdShCtr.

The current thread T first replays the happens-before edge from the last reader thread, by waiting for it to reach the same point it reached during RECORD. In Figure 1, T3 replays the edge from T2 by waiting for T2 to perform the same number of conflicting transitions involving a read (i.e., transitions to RdEx_{T2}), which guarantees that T3's access transitively happens after the last write (by T1).

Next, T waits for all prior gRdShCtr increments to occur, since gRdShCtr increments are globally ordered. T waits until gRdShCtr equals T.log.nextRdShCtr - 1; then it increments gRdShCtr to T.log.nextRdShCtr.

Fence transitions. To replay a fence transition on an object in RdSh_{c} state, a thread needs to replay the happens-before edge from the prior transition to RdSh_{c}. Thread T's instrumentation slow path uses the following pseudocode to wait for gRdShCtr to reach the expected value:

```
case FENCE:
  while (gRdShCtr < T.log.nextRdShCtr) {
    memory_fence;
  }
  memory_fence;
  T.log.readNextEntry();
  break;
```

where T.log.nextRdShCtr is the recorded value c (from RdSh_{c}). In Figure 1, T4 replays a fence transition by waiting for the global counter gRdShCtr to reach c. Similarly, T5 replays a fence transition by waiting for gRdShCtr to reach c+1.

5.2 Eliding Program Synchronization Operations

Unlike most other record & replay systems, our RECORD and REPLAY analyses do *not* record and replay the order of synchronization operations such as lock acquire and release, monitor wait and notify, and thread fork and join. By replaying all recorded cross-thread *data* dependences, REPLAY preserves the same behavior as the recorded execution. (We note that programs use synchronization in order to limit shared-memory interleavings and provide properties such as atomicity, ordering, and visibility. By replaying data dependences faithfully, REPLAY naturally preserves these properties.)

Not only can REPLAY avoid replaying synchronization operations, but it can *elide* synchronization operations. That is, during REPLAY, REPLAY can compile synchronization operations as no-ops. In fact, REPLAY *must* elide synchronization operations in order to avoid deadlock. Consider the example in Figure 2. Suppose during RECORD, T1 acquires m's lock first, so T1's store to o.f occurs before T2's load, and RECORD records a happens-before edge from a safe point after T1's critical section to T2's load. If REPLAY allowed synchronization operations to execute (without recording and replaying synchronization operations), then T2 could acquire

```
// T1:                    // T2:
synchronized (m) {        synchronized (m) {
  ...                       ...
  o.f = ...;                ... = o.f;
  ...                       ...
}                         }
```

Figure 2. An example cross-thread data dependence.

m's lock first, making it impossible to replay the write–read dependence on o. In that case, REPLAY would deadlock: T2's load would wait on T1's store, while T1 would wait to acquire m.

In contrast, prior work typically records and replays synchronization operations (Section 8). RECORD and REPLAY *could* record and replay the order of synchronization operations in addition to cross-thread data dependences—incurring additional, unnecessary overhead.

Interestingly, by eliding synchronization operations, the replayed execution can achieve better scalability than a normal execution that includes synchronization. For example, two critical sections that acquire the same lock but do not have a data dependence between them, can execute in parallel in the replayed execution. Thus, the replayed execution often outperforms the recorded execution, and it sometimes even outperforms *baseline* (uninstrumented) execution (Section 7.3).

5.3 Soundness of REPLAY

The correctness of record & replay relies on the observation that value determinism (all reads performed by a replayed execution produce the same results as the original recorded execution) is achieved if all dependences in the recorded run are mimicked in the replayed run.

To preserve all dependences between a recorded execution and its replay, it suffices to preserve only cross-thread dependences. Other dependences, which occur entirely on a single thread, will be enforced by the reordering restrictions of the compiler and hardware.

Prior work shows that Octet creates happens-before relationships between all cross-thread dependences in the recorded run [11]. The information logged by RECORD during a recorded run is sufficient to allow REPLAY to deterministically replay the recorded execution:

Theorem 1. *Given logs produced by* RECORD *during an execution,* REPLAY *deterministically replays that execution, preserving all dependences.*

Proof. We proceed by showing that every cross-thread dependence in the recorded execution (hereafter referred to as rec) is respected by the replayed execution (referred to as rep). We need only account for cross-thread dependences that are not transitively implied by other dependences. We consider each type of dependence in turn.

$w_X \to w_Y$ In rec, this dependence is captured by RECORD as a transition from WrEx_X to WrEx_Y, with Y as the requesting thread and X as the responding thread. X's log notes its DPL when X increments its response counter, while Y's log notes the expected value of X's response counter after the increment, rc_x. As X runs rep, it maintains a response counter, rc. When X reaches the dynamic program point where the coordination occurred, it increments rc by 1. When Y reaches the point where it made the request, it compares rc_x to rc. If $rc \geq rc_x$, B can be sure that A has already performed its write, and hence B's write will happen later, preserving the dependence.

$r_X \to w_Y$ In rec, prior to performing w_Y, Y will find obj in either RdEx_X or RdSh_c state. In either case, RECORD initiates coordination between X and Y. This scenario is therefore analogous to that of the previous case, and w_Y will occur after r_X in rep, preserving the dependence.

$w_X \to r_Y$ There are three possible types of RECORD state transitions in rec that might arise due to this dependence. (i) $\mathsf{WrEx}_\mathsf{X} \to \mathsf{RdEx}_\mathsf{Y}$ requires coordination, and would be enforced as in the previous cases. (ii) If $\mathsf{WrEx}_\mathsf{X} \to \mathsf{RdEx}_\mathsf{Z} \to \mathsf{RdSh}_\mathsf{c}$, Y reads obj after some third thread put it into RdEx_Z. In this case, REPLAY uses a similar mechanism as above to ensure that Z has moved past the point where it put obj into RdEx_Z, but using the Z.numConflReads counter instead of the response counter. Note that REPLAY will also cause Y to wait until any prior transitions to RdSh are complete (i.e., transitions to $\mathsf{RdSh}_{\mathsf{c}'}$ where c' < c). (iii) r_Y could happen when obj was already in RdSh_c state. In this case, we note that updates to gRdShCtr are replayed at the correct times, and that during rec, Y would record c in its log before performing r_Y. Before performing r_Y in rep, Y ensures that gRdShCtr is at least the recorded value. This ensures that *all* RdEx $\to$ RdSh transitions that happened before r_Y in rec have happened in rep, again preserving the dependence.

All other dependences in the program are transitively implied by some combination of these cross-thread dependences and intra-thread dependences, which are maintained by the reordering rules of the compiler and hardware. Hence, all dependences in rec are preserved in rep, providing value determinism. □

6. Deterministic Execution

The prior sections described how to record and replay an execution's cross-thread data dependences. However, cross-thread data dependences are not the only source of nondeterminism—particularly for a managed language virtual machine (VM) such as a Java VM (JVM). Language VMs incur significant nondeterminism from features such as dynamic class loading, dynamic optimization, and automatic memory management.

To demonstrate our REPLAY analysis, our goal is to provide *application-level determinism*: the RECORD and REPLAY runs should appear identical from the application's perspective, i.e., the two runs should perform the same loads from memory and get the same values, and the two runs should produce the same output (i.e., perform the same system calls).[3] Application-level determinism is sufficient to handle the existing use cases for online and offline record & replay: reproducing executions at a later time, for debugging or understanding purposes; and running multiple identical copies of processes simultaneous, for fault tolerance or distributed dynamic analysis (Section 1).

To provide application-level determinism, the JVM internally does not need to execute deterministically. However, some nondeterministic behaviors of the JVM affect application behavior that would otherwise be deterministic. To provide application-level determinism, the JVM must control these sources of nondeterminism. We modify the JVM to handle sources of nondeterminism, either by recording and replaying them or by making them inherently deterministic.

To deal with two particularly challenging sources of nondeterminism—dynamic compilation and class loading—we introduce a methodology called *fork-and-recompile*. We call the fork-and-recompile methodology a "research methodology" because it is not suitable for production environments. With substantially more

[3] Note that because Java does not make object addresses available to the application, application-level determinism does not require layout determinism. However, it does require deterministic hash codes (Section 6.3).

engineering effort, we believe a JVM could be modified to control sources of nondeterminism *without* requiring this methodology, making it practical for production settings (Section 6.6).

The nondeterminism problems that this section addresses, and the solutions that we introduce, are largely specific to managed language VMs. Furthermore, some sources of nondeterminism are likely to be relevant only to *metacircular* VMs (VMs written in the same language that they implement) [32] such as Jikes RVM. This section's issues are *not* however limited to our RECORD and REPLAY analyses, nor to their approach for representing dynamic program location. Any system providing deterministic replay or deterministic execution needs to deal with the VM- and system-level sources of nondeterminism that this section tackles.

Since our efforts to provide application-level determinism are largely at the implementation level, we first overview the implementations of RECORD and REPLAY. Subsequent subsections describe challenges and corresponding solutions for handling nondeterminism.

6.1 RECORD and REPLAY Implementation Overview

We have implemented RECORD and REPLAY in Jikes RVM, a high-performance JVM [1]. Although Jikes RVM targets research, its performance is competitive with commercial JVMs [7]. The RECORD implementation builds on the publicly available Octet implementation [11]. Our implementations of RECORD and REPLAY are publicly available on the Jikes RVM Research Archive.[4]

Instrumentation. RECORD and REPLAY modify Jikes RVM's dynamic compilers to instrument all application and Java library code. However, Jikes RVM is itself written in Java, so the JVM and the application call into the same Java libraries. RECORD and REPLAY need to instrument libraries called from the application, but not libraries called from the JVM. To accomplish this, we have modified the dynamic compilers to produce two versions of each library method: one for application context and one for JVM context.

Maintaining DPL. During RECORD and REPLAY, each thread T maintains its current DPL by incrementing T.dynCtr at every method entry, method return, and loop back edge. During RECORD, at every program point in the application that is a potential safe point—meaning it might call into the VM and record a happens-before source—instrumentation stores the current static site in a per-thread variable. Combining this static site with T.dynCtr allows RECORD to compute an application DPL even though the coordination response can occur within nested calls to the VM. REPLAY executes instrumentation at every safe point that checks whether the current DPL matches the next event in the current thread's log.

Eliding synchronization operations. REPLAY modifies the dynamic compilers to ignore lock acquire and release operations (corresponding to synchronized blocks in the original Java code). REPLAY modifies the VM's implementations of Object.wait(), Object.notify(), and Object.notifyAll() so that they have no effect if the object's lock is not held.

Instrumenting special accesses. The application performs some memory accesses by calling into the VM to perform the accesses. Examples are System.arraycopy() and Object.clone(), which call into the VM; I/O routines that read or write buffers; and calls from native code that access the Java heap. We have identified these cases and added explicit instrumentation so that RECORD and REPLAY perform appropriate checks before these memory accesses.

[4] `http://www.jikesrvm.org/Resources/ResearchArchive/`

6.2 Dynamic Compilation and Class Loading

Challenges of dynamic compilation. A compiled method may be recompiled multiple times at different optimization levels. Method inlining leads to different control flow, affecting the frequency of T.dynCtr increments—and thus computing DPL nondeterministically. Optimizations may eliminate redundant loads and stores nondeterministically.

Optimization decisions depend on timer-based sampling, and optimized compilation is by default performed concurrently with program execution [3], so optimization and execution of optimized code are inherently nondeterministic from run to run. A production implementation could, in theory, record and replay optimization decisions. Jikes RVM does *not* provide such support. (We note that Jikes RVM *does* support a methodology called *replay compilation* that records some compilation decisions and profile information, called "advice," to make compilation decisions somewhat deterministic in a run that uses the advice [23]. However, compiled code is not deterministic between runs that *generate* and *use* the advice. We thus do not use replay compilation.)

Challenges of dynamic class loading and initialization. When a class is first accessed, the accessing thread triggers class loading and initialization of the class. *Which* thread is first is nondeterministic; application-level replay of cross-thread dependences does *not* make class loading and initialization deterministic automatically. Initializing a class involves calling the class's static initializer (static variable initialization and static $\{\ldots\}$ code blocks), which is *application* code that must be executed by the same thread during RECORD and REPLAY in order to provide determinism. Although triggering of class initializers can, in theory, be recorded and replayed, a more difficult problem is making *custom* class loaders (class loaders that call code provided by the application) deterministic. When a metacircular VM performs custom class loading, the activities of the application and VM are tightly coupled, and accounting for this coupling is difficult: application-context code elides synchronization operations, but VM-context code perform synchronization operations, leading to deadlocks that we have been unable to avoid without compromising determinism.

Solutions. A production-quality implementation could address the challenges of nondeterministic dynamic compilation, class initialization, and custom class loading by carefully recording and replaying these behaviors. Instead, we introduce a methodology called *fork-and-recompile* to sidestep these challenges and demonstrate the correctness and performance of RECORD and REPLAY.

When using fork-and-recompile methodology, the JVM executes *two* iterations of the program. The sole purpose of the first, "warmup" iteration of the program is to compile all the code, and load and initialize all classes. While these behaviors are nondeterministic, both RECORD and REPLAY runs will start from the *same state* after the warmup iteration finishes.

After the warmup iteration finishes, the JVM forks its process using the fork system call. Since fork on Linux works correctly only if the process has a single thread, our implementation first forces all threads—any remaining application threads, as well as JVM threads that perform compilation, GC, and profiling—except the main thread to terminate. After fork returns, we designate the child process as the RECORD process and the parent process as the REPLAY process. The REPLAY process waits for the RECORD process to complete. After the RECORD process completes, the REPLAY process proceeds. With additional engineering effort, it should be possible to run RECORD and REPLAY simultaneously, with REPLAY reading directly from RECORD's logs, demonstrating *online* replay.

The RECORD and REPLAY processes each first recompile all application-context methods with RECORD or REPLAY instrumentation, respectively. They recompile each method using the same optimizations from the warmup iteration, providing a realistic mix of optimized and unoptimized code for evaluating performance. The RECORD and REPLAY processes thus start from the same state in terms of dynamic compilation and loaded and initialized classes.

6.3 Nondeterminism Caused by Garbage Collection

Challenges. Jikes RVM's default high-performance garbage collector (GC) is stop-the-world, parallel, and generational [9]. When a thread's allocation triggers GC, all threads stop at GC-safe points (periodic program points where GC can happen safely). Then multiple GC threads perform either a nursery collection (which collects only recently allocated objects) or a full-heap collection. *When* GC is triggered is nondeterministic, even if the application executes deterministically, because the JVM allocates objects into the same heap. Furthermore, stopping each thread at a GC-safe point is inherently racy and nondeterministic.

In theory, triggering GC nondeterministically might not affect application-level determinism, but in practice, it presents several challenges. Nondeterministic GC leads to different behaviors for weak references and finalizers, whose behavior depends on when dead objects are collected. Furthermore, low-level I/O routines in Jikes RVM behave differently depending on whether buffer arrays, provided as parameters, are in moving or non-moving spaces (since the routines must operate on unmovable buffer objects).

Not only is nondeterministic triggering of GC problematic, but so is the the operation of GC itself, which moves objects nondeterministically during collection, leading to nondeterministic object addresses. Furthermore, the addresses of newly allocated objects are nondeterministic due to VM allocation nondeterminism and GC nondeterminism. In theory, object addresses should not affect application-level behavior since object addresses are not visible to the application. However, in Jikes RVM and other JVMs, the default implementation of Object.hashCode() (i.e., the "identity" hash code) returns a value based on the object's address.[5] Nondeterministic hash code values can preclude application determinism; for example, they affect layout and iteration order in a hash table whose key objects use the identity hash code.

Solutions. To avoid conflicting with other replayed happens-before edges and to avoid nondeterministic behavior of weak references, finalizers, and I/O routines, the implementation records and replays GC points. During RECORD, each thread that triggers or joins a collection, records the event in its log: the DPL and whether the collection was nursery or full-heap. During REPLAY, threads perform GC at the recorded DPLs; they cannot trigger GC otherwise.

To support application-level determinism, we allow object addresses to be nondeterministic but make hash codes deterministic. RECORD and REPLAY instrument each object allocation so it sets a dedicated header word to a deterministic encoding of the current thread T and dynamic counter T.dynCtr. The identity hash code operation Object.hashCode() reads from and returns this value.

Replaying GC points deterministically could, in theory, lead to deterministic finalizer behavior. However, parallel GC identifies finalizable objects in a nondeterministic order, so the finalizer thread calls the objects' finalize() methods in nondeterministic order. While the implementation could potentially enforce a deterministic order to calling the finalizers, for simplicity our implementation currently *disables* finalizers. This choice, while not appropriate for a production implementation, does not necessarily violate semantics, since the JVM specification does not mandate timely or eventual calling of finalizers [33].

6.4 Other Nondeterministic System Behavior

Challenges. Querying the current system time is inherently nondeterministic and can affect application behavior nondeterministically. Nondeterministic I/O is similarly problematic, e.g., for interactive applications or network I/O.

Solutions. Recording and replaying the value of system time could be expensive if the application queries it often. Instead, our implementation simulates deterministic time by keeping track of a counter that represents "logical time." It increments logical time whenever time is queried. Although time is conceptually a global value, for simplicity our implementation uses per-thread counters. This approach will not work well if a thread compares a time value from another thread or if time values need to correspond better to actual time values. However, it works well enough for the programs we evaluate.

Our implementation and experiments avoid most challenges of nondeterministic I/O. We evaluate benchmarked programs that read and write the file system deterministically, without reading from nondeterministic I/O sources such as the console or network. The implementation provides RECORD and REPLAY with the same initial directory structure by backing up the current directory's contents before RECORD starts, and restoring the same before REPLAY starts. A production implementation would need to record and replay nondeterministic I/O sources and other system behavior such as the side effects of system calls. In contrast, our goal is to provide the minimum system- and JVM-level determinism needed to achieve application-level determinism for our evaluated programs, in order to demonstrate and evaluate RECORD and REPLAY.

6.5 Verifying Determinism

Even if an execution replays successfully, how do we know that its application-level behavior is the same as during the recorded execution? We define "same behavior" as meaning that the program performed the same loads from memory and got the same values. The RECORD and REPLAY implementations support *value logging* configurations that record (during RECORD) and check (during REPLAY) the value of every program load and store—or a hash of the last k values—ensuring that REPLAY's enforcement of happens-before edges is sufficient to produce value determinism.

The program should also perform the same system calls, e.g., output to the console. To ensure system call determinism, we rely on each benchmark's harness, which validates the contents of the console output and output files.

6.6 Discussion: Making Determinism Practical

With significantly more effort, it should be possible to provide determinism using standard "adaptive" methodology. A key challenge for our implementation is that Jikes RVM itself is written in Java and shares many components with the application, including the heap, the adaptive optimization subsystem, and the libraries. Cleanly separating the JVM and application would make it substantially easier and cheaper to provide application-level determinism. For example, application object addresses could be made deterministic, automatically providing deterministic hash codes. Compilation and class loading decisions could be recorded and replayed in a way that would not interact poorly with the JVM.

7. Evaluation

This section evaluates the effectiveness and efficiency of RECORD and REPLAY using our JVM determinism modifications and fork-and-recompile methodology.

[5] If GC moves an object whose Object.hashCode() method has been called, GC annotates the moved object with its old address, so that Object.hashCode() continues to return the same value.

	Threads		Transitions triggered by memory accesses				Log
	All	Live	None	Conflicting	Upgrading*	Fence	MB/s
hsqldb6	402	102	6.9×10^{8}	9.1×10^{5}	1.3×10^{5}	6.3×10^{4}	0.7
lusearch6	65	65	2.7×10^{9}	4.8×10^{3}	1.9×10^{2}	2.5×10^{3}	<0.1
xalan6	9	9	1.1×10^{10}	1.8×10^{7}	2.3×10^{3}	4.3×10^{3}	7.7
avrora9	27	27	5.8×10^{9}	5.9×10^{6}	8.9×10^{5}	4.4×10^{6}	2.5
jython9	3	3	7.3×10^{9}	5.1×10^{1}	0	0	<0.1
luindex9	2	2	3.7×10^{8}	4.6×10^{2}	5.2×10^{1}	1.0×10^{0}	<0.1
lusearch9	64	64	2.6×10^{9}	3.7×10^{3}	7.9×10^{2}	6.8×10^{3}	<0.1
pmd9	5	5	7.2×10^{8}	5.4×10^{4}	6.8×10^{3}	2.0×10^{4}	0.1
sunflow9	128	64	1.7×10^{10}	7.2×10^{3}	5.2×10^{3}	2.1×10^{4}	0.1
xalan9	32	32	1.1×10^{10}	2.0×10^{7}	1.4×10^{4}	6.0×10^{4}	9.7
pjbb2000	37	9	2.1×10^{9}	1.2×10^{6}	3.1×10^{5}	3.1×10^{3}	1.1
pjbb2005	9	9	7.8×10^{9}	4.6×10^{7}	6.4×10^{6}	1.5×10^{7}	4.8

Table 2. Characteristics of recorded and replayed executions, based on a statistics-gathering configuration of Record. *Upgrading transitions exclude $\mathsf{RdEx_T}\rightarrow\mathsf{WrEx_T}$ transitions since Record does not record happens-before edges for them (Table 1).

7.1 Methodology

Benchmarks. Our experiments execute benchmarked versions of large, real-world applications:

- the DaCapo benchmarks [8], versions 2006-10-MR2 and bach-9.12 (distinguished with suffixes 6 and 9), excluding benchmarks that are single-threaded or that (unmodified) Jikes RVM cannot execute correctly
- fixed-workload versions of SPECjbb2000 and SPECjbb2005[6]

We exclude eclipse6 from most experiments because (1) it fails with the fork-and-recompile methodology (we are unable to restart eclipse6's worker threads correctly), and (2) runtime support for determinism causes eclipse6 to execute incorrectly.

Experimental setup. To account for run-to-run variability (due to dynamic optimization guided by timer-based sampling) and any machine noise, each performance result is the median of 15 trials. We also show the mean, as the center of 95% confidence intervals. We build and use a high-performance configuration of Jikes RVM that adaptively optimizes the application as it runs. We let Jikes RVM's default high-performance garbage collector adjust the heap size automatically.

Platform. The experiments execute on an AMD Opteron 6272 system with eight 2 GHz 8-core processors (64 cores total), running 64-bit RedHat Enterprise Linux 6.5, kernel 2.6.32.

We execute xalan9 using only 32 cores since it shows anomalous performance using 64 cores: for xalan9 on 64 cores, all Record and Replay configurations outperform the baseline (unmodified JVM). Other researchers have observed that such anomalies are an artifact of Linux thread scheduling decisions [6].

7.2 Record & Replay Characteristics

Table 2 shows statistics for recorded and replayed executions. Of 12 evaluated programs, 9 have at least 8 simultaneously live threads, and 5 programs have at least 32 live threads. The next four columns show that most accesses do not trigger a state transition, thus requiring no logging. Programs with a higher rate of accesses that trigger state transitions, incur higher costs (Section 7.3). The last column of the table shows that programs with a higher rate of state transitions lead to a higher rate of logging events, although the rate does not exceed 10 megabytes per second.

The first column of Table 3 reports how frequently Replay successfully provides deterministic replay. Each percentage is the

	Replay default	Ignore HB edges	Keep sync.
hsqldb6	100%	0%	0%
lusearch6	100%	0%	0%
xalan6	100%	0%	0%
avrora9	100%	0%	0%
jython9	100%	0%	80%
luindex9	100%	0%	0%
lusearch9	100%	0%	0%
pmd9	100%	0%	0%
sunflow9	100%	0%	0%
xalan9	60%	0%	0%
pjbb2000	100%	0%	0%
pjbb2005	100%	0%	0%

Table 3. Percentage of five executions replayed successfully, for various Replay configurations.

number of successfully replayed trials, out of five trials, using the fork-and-recompile methodology. We enable value logging in order to ensure that successfully replayed runs execute deterministically (i.e., load the same value at each load). For most programs, the default Replay configuration consistently replays recorded executions successfully. For xalan9, Replay sometimes fails. These failures are due to limitations of our implementation: sources of nondeterminism that we have been unable to identify and address. (Replay occasionally fails for a few other programs. We ran additional trials to verify that with high confidence, the expected number of successful trials, out of five, is closer to five than four.)

How do we know Replay is actually doing anything important? That is, is it actually necessary to record and replay cross-thread dependences in order to replay these programs deterministically? The *Ignore HB edges* configuration *ignores* recorded happens-before edges during replay, but enables value logging. All executions fail, either with value logging errors or other program errors. The *Keep sync.* configuration performs synchronization during replay, instead of eliding synchronization. All programs except jython9, which has little multithreaded behavior, deadlock consistently because they have conflicting critical sections that interleave nondeterministically, similar to Figure 2. These configurations' failures demonstrate that replaying these programs deterministically does not happen serendipitously. Instead, deterministic replay requires recording and replaying cross-thread dependences accurately, and eliding synchronization operations during Replay.

7.3 Performance

This section first evaluates Record by executing the JVM using the default "adaptive" methodology in which the JVM recompiles and optimizes methods at run time using online, sampling-based

[6] `http://www.spec.org/jbb200{0,5}`, `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`

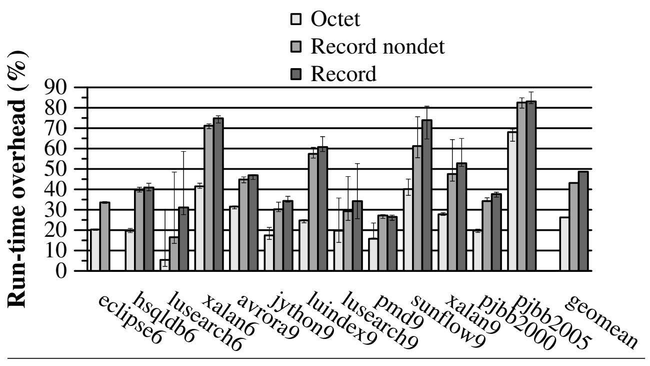

Figure 3. Performance of RECORD and two partial RECORD configurations using default adaptive methodology. The graph excludes RECORD performance for eclipse6 because Jikes RVM with our determinism modifications fails to execute eclipse6 correctly (Section 7.1).

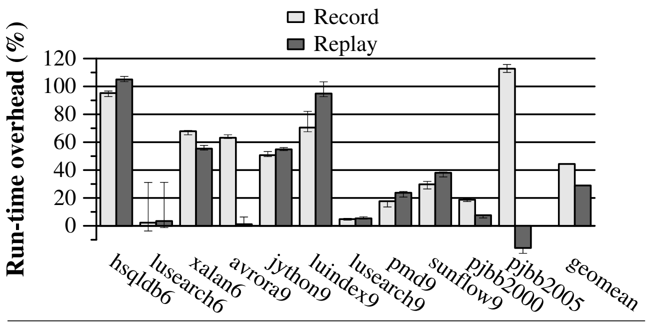

Figure 4. RECORD and REPLAY performance using fork-and-recompile methodology.

profiling. It then evaluates RECORD and REPLAY using our fork-and-recompile methodology.

Adaptive methodology. Figure 3 shows the overhead that RECORD adds to programs when using adaptive methodology. Each bar is the run-time overhead over unmodified Jikes RVM. The first bar is the overhead of using *Octet* to track cross-thread dependences [11]. Octet adds 26% overhead on average. Programs with a higher rate of state transitions—especially conflicting transitions (Table 2)—incur more overhead. The program with the highest overhead, pjbb2005, also has the highest rate of conflicting transitions. High-conflict programs are a general concern for the RECORD analysis and other analyses built on top of optimistic tracking of dependences. Addressing this issue is beyond the scope of this paper, but we note that our recent work develops a *hybrid* of optimistic and pessimistic tracking that adaptively applies pessimistic tracking to high-conflict objects, reducing overhead substantially for high-conflict programs without significantly impacting low-conflict programs [14].

Record nondet includes the additional costs to record happens-before edges identified by Octet. These costs include writing happens-before sources and sinks to per-thread logs on disk, incrementing T.dynCtr to maintain DPL, and setting the last application site ID at every potential safe point that might call into the JVM. This nondeterministic RECORD configuration adds an additional 17% overhead (relative to baseline execution) over Octet, and 43% overall.

Finally, *Record* is the full default RECORD configuration. It adds support for making some JVM features deterministic, such as making GC, hash codes, and system time deterministic. These features have their costs, e.g., deterministic hash codes must be initialized at allocation time, but together they add modest overhead: 6% (relative to baseline execution) on average over nondeterministic RECORD. Overall, RECORD slows program execution by 49% on average.

These RECORD results serve as a comparison against the following results, which use the nonstandard fork-and-recompile methodology but support both RECORD and REPLAY.

Fork-and-recompile methodology. Figure 4 shows the overhead RECORD and REPLAY add over baseline (unmodified JVM) execution. All configurations, including the baseline, use the fork-and-recompile methodology described in Section 6.2.

Record is the default recording configuration; it is the same as the *Record* configuration from Figure 3. With fork-and-recompile methodology, RECORD adds 44% overhead on average, which is comparable to the 49% overhead using adaptive methodology (Figure 3). For some programs, the overhead of RECORD is substantially different between the two methodologies. We find that differences are due to the fact that adaptive execution includes compilation time and spends time executing unoptimized code before optimizing it, whereas fork-and-recompile methodology does not. Shorter-running programs experience this effect more acutely than longer-running programs.

Replay is the full default REPLAY analysis: it replays cross-thread dependences, tracks DPL by updating T.dynCtr, and includes the determinism changes used by RECORD. We exclude xalan9 because it almost never replays successfully *without* value logging, although it often replays successfully *with* value logging (Table 3), since its nondeterministic failures are timing sensitive.

Interestingly, REPLAY often outperforms RECORD. REPLAY can provide lower overhead than RECORD because it is cheaper to replay known dependences than to record unknown dependences. In particular, RECORD requires that threads coordinate for a conflicting transition, but replaying a conflicting transition's happens-before edge requires no synchronization except memory fences.

REPLAY not only often does less work than RECORD, but REPLAY also enables more parallelism than both RECORD and baseline execution. By eliding program synchronization, REPLAY allows code protected by critical sections and other synchronization to overlap. For example, in Figure 2 (page 6), both threads can enter their critical sections at the same time. The benchmark pjbb2005 uses coarse-grained synchronization that is more conservative than the actual cross-thread data dependences. By eliding synchronization and replaying only the cross-thread data dependences, REPLAY *outperforms baseline (unmodified JVM) execution*.

On average, REPLAY's run-time overhead is 29%. The overheads of RECORD and REPLAY compare favorably with the reported overheads for other approaches that provide both online and offline replay for racy executions on commodity systems [30, 48]. The highest-performing techniques that satisfy these criteria are DoublePlay and Chimera [30, 48]; DoublePlay's overhead is over 100% (unless twice as many cores are used), and Chimera's overhead for CPU-bound programs is 86% (Section 8.1).

8. Related Work

This section covers existing record & replay approaches, alternatives to record & replay, and work that handles nondeterminism in language VMs.

8.1 Record & Replay

Ignoring data races. *RecPlay* and *JaRec* record only high-level races (races between synchronization operations), by recording dependences among synchronization operations such as lock acquire–

release and monitor wait–notify [22, 46]. However, these approaches are *unsound* for executions with data races.

Is it practical to detect or eliminate all data races? Data races are difficult to eliminate; programmers accidentally or intentionally introduce data races when trying to minimize synchronization costs. Detecting or eliminating data races is a well-studied problem (e.g., [12, 21, 37]). Even with recent advances, dynamic approaches slow programs by about an order of magnitude [21]. Static analysis can identify definitely data-race-free (DRF) accesses, but sound static analysis reports many false positives for large programs (e.g., [37, 38, 50]). Dynamic analysis can avoid instrumenting these definitely DRF accesses, but instrumenting the remaining accesses still slows programs substantially [15, 30, 49].

Avoiding explicit tracking of racy accesses. Several approaches avoid explicitly detecting and recording the dependences between racy accesses. *Respec* supports *online* replay by recording synchronization operations and speculating that most data races do not lead to external effects, but cannot provide *offline* replay without additional support such as probabilistic search [31]. *DoublePlay* supports both online and offline replay [48], but needs twice the number of cores to achieve low overhead. Without extra cores, DoublePlay adds 100% overhead.

Other approaches offer probabilistic offline replay based on reproducing executions from limited recorded information, but do not support online replay [2, 26, 34, 42, 51].

Tracking racy interleavings. Our approach records and replays cross-thread data dependences explicitly. Prior work has also tracked cross-thread data dependences, with various limitations.

Instant Replay and *ORDER* track dependences in racy executions by instrumenting all memory accesses, slowing executions by about an order of magnitude [29, 53]. *SMP-ReVirt* and *Scribe* use memory page protection to trigger hardware traps at potentially conflicting accesses [20, 27]. Detecting sharing at page granularity with the expense of page faults is unlikely to scale well to applications with significant true or false sharing. The Scribe authors report low overheads for applications such as web servers and web browsers that may not incur much shared-memory communication [27]; it would be revealing to see how Scribe performs on shared-memory-intensive applications such as PARSEC and SPLASH-2. (Admittedly, our evaluation, which targets the Java platform, also does not evaluate on PARSEC or SPLASH-2, which are C/C++ programs.)

Chimera uses whole-program static data race detection to eliminate instrumentation at definitely data-race-free accesses [30]. However, the remaining instrumentation still slows programs by more than an order of magnitude. Chimera reduces costs further by converting fine-grained synchronization to coarse-grained synchronization. However, this lock coarsening relies on profiling to identify low-conflict regions suitable for the optimization. Chimera slows programs by 86% on average for CPU-intensive benchmarks. Our approach is complementary to Chimera's and could potentially be combined with it.

Light reduces run-time overhead by recording only cross-thread *write–read* dependences [34]. As a result, Light requires offline constraint solving to replay an execution, so it supports only offline replay [34].

Prähofer et al. present an approach for recording and replaying hard real-time programs [44]. The approach records an execution by observing the order of inter-thread data dependences. It replays an execution offline by reconstructing the order of real-time tasks from the data dependences.

Custom hardware support can achieve low-overhead record & replay by piggybacking on cache coherence protocols [24, 25, 36, 39, 43, 52]. However, manufacturers have been reluctant to add complexity to already-complex coherence protocols.

8.2 Deterministic Execution

Run-time determinism. An alternative to record & replay is executing multithreaded programs *deterministically* [5, 18, 19, 35, 41]. Runtime determinism approaches face challenges similar to those for record & replay. They either do not handle racy programs [41], add high overhead [5, 18], or require custom hardware [19]. *Dthreads* provides determinism by mapping threads to processes in order to provide separate address spaces for each thread, which it merges at each synchronization point [35]. Its approach will not scale well to programs that use a lot of fine-grained synchronization or cross-thread sharing of pages.

Language determinism. New languages can provide determinism at the language level [10, 45]. *Determinator* provides determinism with support from the programming model and operating system [4]. Using these approaches requires rewriting programs.

8.3 Making JVMs Deterministic

Prior work has made JVMs deterministic through a combination of controlling and recording nondeterministic behavior [16, 22, 53]. These approaches either modify JVMs written in C/C++ [16, 53] or use dynamic bytecode rewriting [22]. Our implementation faces a unique challenge: it targets a JVM written in Java that shares components with the application.

Rather than producing a production-ready approach, our goal is to evaluate the RECORD and REPLAY analyses, so we introduce a methodology and JVM modifications that control most sources of JVM nondeterminism. Prior work called *Ditto* also modifies Jikes RVM, but its support for deterministic replay appears to be quite limited: it can replay microbenchmarks only, not real programs [47].

9. Conclusion

Our RECORD and REPLAY analyses demonstrate how to record and replay cross-thread dependences efficiently. To evaluate REPLAY, we have modified a JVM and introduced a methodology to support application-level determinism. Our evaluation shows that RECORD and REPLAY outperform competing approaches that target commodity systems, suggesting that this work represents an advance in the state of the art for achieving practical multithreaded record & replay in production systems.

A limitation of this paper's solution to JVM nondeterminism is that the fork-and-recompile methodology is not well suited to most production settings. Future work could modify a JVM in order to limit and control sources of nondeterminism, in order to provide efficient multithreaded record & replay in production settings.

Acknowledgments

We thank Swarnendu Biswas, Brian Demsky, Joe Devietti, Madan Musuvathi, Aritra Sengupta, and Minjia Zhang for helpful ideas and other feedback. Thanks to the anonymous reviewers for insightful suggestions on the text.

References

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. *IBM Systems Journal*, 44:399–417, 2005.

[2] G. Altekar and I. Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *SOSP*, pages 193–206, 2009.

[3] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *OOPSLA*, pages 47–65, 2000.

[4] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient System-Enforced Deterministic Parallelism. In *OSDI*, pages 1–16, 2010.

[5] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *ASPLOS*, pages 53–64, 2010.

[6] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. In *OOPSLA*, pages 81–96, 2009.

[7] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia. Valor: Efficient, Software-Only Region Conflict Exceptions. In *OOPSLA*, 2015. To appear.

[8] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*, pages 169–190, 2006.

[9] S. M. Blackburn and K. S. McKinley. Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *PLDI*, pages 22–32, 2008.

[10] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel Programming Must Be Deterministic by Default. In *HotPar*, pages 4–9, 2009.

[11] M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. Fathi Salmi, S. Biswas, A. Sengupta, and J. Huang. Octet: Capturing and Controlling Cross-Thread Dependences Efficiently. In *OOPSLA*, pages 693–712, 2013.

[12] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, pages 211–230, 2002.

[13] T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault Tolerance. In *SOSP*, pages 1–11, 1995.

[14] M. Cao, M. Zhang, and M. D. Bond. Drinking from Both Glasses: Adaptively Combining Pessimistic and Optimistic Synchronization for Efficient Parallel Runtime Support. In *WoDet*, 2014.

[15] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *PLDI*, pages 258–269, 2002.

[16] J.-D. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *SPDT*, pages 48–59, 1998.

[17] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *USENIX*, pages 1–14, 2008.

[18] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable Deterministic Multithreading Through Schedule Memoization. In *OSDI*, pages 1–13, 2010.

[19] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *ASPLOS*, pages 85–96, 2009.

[20] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution Replay of Multiprocessor Virtual Machines. In *VEE*, pages 121–130, 2008.

[21] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, pages 121–133, 2009.

[22] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. JaRec: A Portable Record/Replay Environment for Multi-threaded Java Applications. *SPE*, 34(6):523–547, 2004.

[23] A. Georges, L. Eeckhout, and D. Buytaert. Java Performance Evaluation through Rigorous Replay Compilation. In *OOPSLA*, pages 367–384, 2008.

[24] D. R. Hower and M. D. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *ISCA*, pages 265–276, 2008.

[25] D. R. Hower, P. Montesinos, L. Ceze, M. D. Hill, and J. Torrellas. Two Hardware-Based Approaches for Deterministic Multiprocessor Replay. *CACM*, 52:93–100, 2009.

[26] J. Huang, C. Zhang, and J. Dolby. CLAP: Recording Local Executions to Reproduce Concurrency Failures. In *PLDI*, pages 141–152, 2013.

[27] O. Laadan, N. Viennot, and J. Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *SIGMETRICS*, pages 155–166, 2010.

[28] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21(7):558–565, 1978.

[29] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE TOC*, 36:471–482, 1987.

[30] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: Hybrid Program Analysis for Determinism. In *PLDI*, pages 463–474, 2012.

[31] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism. In *ASPLOS*, pages 77–90, 2010.

[32] Y. Lin, S. M. Blackburn, and D. Frampton. Unpicking The Knot: Teasing Apart VM/Application Interdependencies. In *VEE*, pages 181–190, 2012.

[33] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Prentice Hall PTR, 2nd edition, 1999.

[34] P. Liu, X. Zhang, O. Tripp, and Y. Zheng. Light: Replay via Tightly Bounded Recording. In *PLDI*, pages 55–64, 2015.

[35] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient Deterministic Multithreading. In *SOSP*, pages 327–336, 2011.

[36] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *ISCA*, pages 289–300, 2008.

[37] M. Naik and A. Aiken. Conditional Must Not Aliasing for Static Race Detection. In *POPL*, pages 327–338, 2007.

[38] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. In *PLDI*, pages 308–319, 2006.

[39] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies Using Strata. In *ASPLOS*, pages 229–240, 2006.

[40] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing Security Checks on Commodity Hardware. In *ASPLOS*, pages 308–318, 2008.

[41] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, pages 97–108, 2009.

[42] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *SOSP*, pages 177–192, 2009.

[43] G. Pokam, C. Pereira, K. Danne, R. Kassa, and A.-R. Adl-Tabatabai. Architecting a Chunk-based Memory Race Recorder in Modern CMPs. In *MICRO*, pages 576–585, 2009.

[44] H. Prähofer, R. Schatz, C. Wirth, and H. Mössenböck. A Comprehensive Solution for Deterministic Replay Debugging of SoftPLC Applications. *IEEE Trans. Industrial Informatics*, 7(4):641–651, 2011.

[45] M. C. Rinard and M. S. Lam. The Design, Implementation, and Evaluation of Jade. *TOPLAS*, 20:483–545, 1998.

[46] M. Ronsse and K. De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *TOCS*, 17:133–152, 1999.

[47] J. M. Silva, J. Simão, and L. Veiga. Ditto – Deterministic Execution Replayability-as-a-Service for Java VM on Multiprocessors. In *Middleware*, pages 405–424, 2013.

[48] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *ASPLOS*, pages 15–26, 2011.

[49] C. von Praun and T. R. Gross. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. In *PLDI*, pages 115–128, 2003.

[50] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *ESEC/FSE*, pages 205–214, 2007.

[51] D. Weeratunge, X. Zhang, and S. Jagannathan. Analyzing Multicore Dumps to Facilitate Concurrency Bug Reproduction. In *ASPLOS*, pages 155–166, 2010.

[52] M. Xu, R. Bodik, and M. D. Hill. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *ISCA*, pages 122–135, 2003.

[53] Z. Yang, M. Yang, L. Xu, H. Chen, and B. Zang. ORDER: Object Centric Deterministic Replay for Java. In *USENIX*, pages 30–30, 2011.