Tracking Conflicting Accesses Efficiently for Software Record and Replay

Ohio State CSE Technical Report OSU-CISRC-2/12-TR01
February 2012

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

Milind Kulkarni
Purdue University
milind@purdue.edu

Abstract

Record and replay, which records a multithreaded program's execution in one run and reproduces it deterministically in a second run, is useful for program debugging, fault detection and analysis. The key challenge in multithreaded record and replay is ensuring that conflicting, cross-thread accesses to shared variables are properly detected, recorded and reproduced. Numerous solutions have been proposed in both hardware and software to track these cross-thread accesses, but to date all general-purpose software solutions suffer from high overhead or have serious limitations. This paper introduces ROCTET, an approach for performing software-only deterministic record and replay. ROCTET is built on top of a novel dynamic analysis infrastructure, OCTET, which can detect at low overhead any cross-thread dependences during execution by exploiting the fact that most accesses, even of shared objects, are not part of cross-thread dependences. We implement OCTET and ROCTET in a JVM and show they add low overhead for several multithreaded applications. We also show that our implementation of ROCTET can successfully replay recorded executions when it can successfully control or ignore extraneous sources of nondeterminism in the VM, libraries, and system.

1. Introduction

Finding bugs in multithreaded programs is a notoriously difficult problem, largely due to the non-deterministic interactions of threads accessing shared data: the behavior of a program run that causes a bug to manifest may not be observed during later runs while tracking down the bug. *Offline record and replay* has been proposed to aid this process. While a program runs, its behavior, including non-deterministic interactions, is *recorded*; this behavior can then be deterministically *replayed*. Any bug that appeared in the recorded run will be reproduced in the replayed run. Similarly, *online* record and replay executes the second, replayed run concurrently with the recorded run. This approach is useful for fault tolerance or running dynamic analyses alongside an execution.

Performing record and replay is fairly easy for single-threaded code: the sources of non-determinism are typically readily identifiable and infrequent enough that instrumentation overhead is not a concern. Record and replay is much more difficult for multithreaded code, due to the interactions between threads accessing shared memory. Nondeterministic interleavings occur due to high-level races (i.e., races on synchronization variables) and data races (unsynchronized, conflicting reads and writes). It is relatively inexpensive to record high-level races because synchronization operations are typically infrequent, but any read or write may potentially be involved in data race. Faithfully recording and replaying multithreaded programs requires carefully detecting and managing *conflicting* accesses by threads to shared data.

Numerous existing approaches provide multithreaded record and replay. *Uniprocessor* approaches replay conflicting accesses by scheduling threads identically between the record and replay runs. Unfortunately, such approaches do not scale. *Multiprocessor* record and replay can be supported efficiently in hardware, by monitoring cache-coherence transactions to detect and record conflicting accesses to shared memory. Replay is supported by reproducing those conflicts [21, 22, 31, 33, 51].

In the absence of hardware support for record and replay, there has been much interest in *software-only* approaches. Unfortunately, all existing approaches suffer drawbacks: high overhead due to expensive instrumentation to detect conflicting accesses [27]; no support for offline replay [28]; or only probabilistic guarantees of replay fidelity [4, 38, 50]. The best-performing fully featured, software-only record and replay system is DoublePlay [46], which operates at low overhead when there are extra cores on which to offload the recording processes, but still incurs 100% overhead when all cores are utilized during execution.

This paper presents ROCTET, a software-only, low-overhead, multiprocessor record and replay system that works for all programs and efficiently supports both online and offline replay.

Outline of approach

If every dependence in a recorded run is faithfully reproduced in the replayed run, every load in the latter run will return the same value as the corresponding load of the former run. Hence, modulo other sources of nondeterminism, the replayed execution will deterministically reproduce the recorded execution. Our system is motivated by the following observation: the only dependences that need to be preserved across runs are *cross-thread* dependences, dependences where one access is performed on one thread and the dependent access is on another. Provided that cross-thread dependences are replayed, all other dependences will be enforced by the compiler and hardware.

According to prior work, detecting all cross-thread dependences requires synchronization at each read and write to detect interactions between threads, incurring overhead proportional to the number of memory accesses. However, we present a dynamic analysis framework that can detect conflicting accesses at a cost proportional not to the number of accesses but to the number of *conflicting* accesses (those that may induce cross-thread dependences). Our analysis leverages a key insight: most objects exhibit some thread locality, and hence most accesses *do not* induce cross-thread dependences, even if the object being accessed is shared.

Our analysis framework, OCTET, associates a state with each object that records the last accessor of the object, and its permissions (read or write). When a thread accesses an object in a compatible state, the access proceeds without synchronization. It is only when a conflicting access is attempted that a potential dependence

is detected and communication is required. OCTET is *optimistic*: it assumes most accesses are not involved in cross-thread dependences, and supports them at low overhead, at the cost of expensive communication when a cross-thread dependence may have occurred. OCTET's protocols ensure that there is a happens-before relationship between any two cross-thread dependent accesses. Section 2 describes OCTET in more detail.

Using OCTET as an underlying analysis, we build a record and replay system called ROCTET, described in Section 3, that records dependences detected by OCTET by noting where the conflicting threads are when a dependence is detected (hence establishing the timing of the dependence). On replay, ROCTET manipulates the execution of threads to ensure that all dependences are replayed in the same order as in the recorded run, preserving the recorded happens-before relationships and providing deterministic replay. Note that while our current implementation of ROCTET correctly avoids nondeterminism due to shared-memory accesses, it does not control for certain other sources of nondeterminism (allocation, nondeterministic runtime compilation, etc.), so our replay environment does not provide determinism for these behaviors (*e.g.*, allocation-address based hash codes will not return the same value).

We implement OCTET and ROCTET as dynamic analyses in a high-performance JVM. Across multithreaded benchmarks from the DaCapo Benchmarks and SPEC JBB2000 [10, 45], we show that OCTET can detect conflicting accesses at low overhead (geometric mean: 28%), and that recording a run with ROCTET adds low additional overhead ($\sim$7%). This is significantly lower than the recording overheads of prior software-based full-featured record and replay systems. We argue that ROCTET can replay executions with overhead similar to the recorded execution. However, because our implementation does not control many sources of nondeterminism beyond shared-memory accesses, it can only replay programs with significant limitations that control or ignore nondeterminism and that add high overhead to both record and replay.

2. Tracking Cross-Thread Dependences

Our approach to efficient software record and replay relies on the ability to accurately, and with low overhead, detect *cross-thread dependences* on shared objects: data dependences involving accesses to same variable by different threads.

Capturing cross-thread dependences will allow us to provide deterministic replay of recorded executions, as described in Section 3. This approach to record and replay has typically been taken by hardware techniques that rely on cache coherence to detect and record conflicting accesses [21, 22, 31, 33, 51]. Unfortunately, doing the same in software has proven difficult. Because any potentially shared memory might be involved in a cross-thread dependence, prior approaches have used synchronization at essentially every read and write [27].

Our approach to detecting cross-thread dependences with low overhead is based on a key insight: *the vast majority of accesses, even to shared objects, are not involved in a cross-thread dependence*. If we can detect efficiently whether an access *cannot* create a cross-thread dependence, we can perform synchronization only when conflicting accesses occur, and dramatically lower the overhead of detecting cross-thread dependences.

To achieve this goal, we associate a thread-locality *state* with each potentially shared object that captures which accesses will not cause cross-thread dependences. Accesses consistent with the object's state proceed without synchronization, while attempted accesses in violation of that state imply that a cross-thread dependence might exist. Such accesses trigger a state change and require synchronization. Hence, our technique's costs are proportional to the number of *conflicting* shared memory accesses in a program, rather than the total number of accesses or shared accesses.

We have built a framework implementing this approach called OCTET (optimistic cross-thread explicit tracking). OCTET is "optimistic" because it assumes that most accesses do not create dependences and supports them at low overhead, at the cost of more expensive synchronization when accesses conflict. OCTET's primary function is to detect potential cross-thread dependences in parallel execution and ensure that a happens-before relationship exists between the dependent accesses (even if the original program performs the accesses in a racy manner).

2.1 OCTET states

A thread-locality state for an object tracked by OCTET captures what accesses can be made by that object without synchronization; that is, which accesses can be made to that object that definitely *do not* create any new cross-thread dependences. The possible OCTET states for an object are:

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

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

RdSh: Read shared. Any thread may read the object without synchronization.

Each object's header stores its state. Note the similarity of the thread-locality states to coherence states in the standard MESI cache-coherence protocol [37]. Modified corresponds to $\mathsf{WrEx_T}$, Exclusive corresponds to $\mathsf{RdEx_T}$, and Shared corresponds to RdSh. Invalid corresponds to the state for threads *other than* T when an object has state $\mathsf{WrEx_T}$ or $\mathsf{RdEx_T}$.

Any access by a thread to an object in an incompatible state implies that there *may* be a cross-thread dependence.[1] If such an access is performed, OCTET performs synchronization to safely change an object's state to one compatible with the access.

Figure 1 shows OCTET's states and the program accesses that cause transitions between them. Some transitions are *conflicting* transitions, as they require coordination with other threads to resolve (as described next), while others are *upgrading* transitions, as they can be done without coordination (though they require synchronization).

2.2 Detecting and managing conflicting accesses

OCTET's instrumentation inserts read- and write-barriers before every access of a potentially shared object to ensure that the access is compatible with the object's state, as shown in Figure 2. Low overhead is achieved because when an access is compatible with the object's state, the barrier overhead is merely the cost of the state check; it is only when a conflicting access is detected that a *slow path*, involving synchronization, is entered to allow the conflicting thread to gain access to the object. What does this slow path entail?

2.2.1 Handling conflicting accesses

When a conflict is detected by an OCTET barrier, the state of the object must be changed so that the conflicting thread may access it. However, the object state cannot simply be changed at will. If thread T changes the state of an object while another thread S that has access to the object is between its state check and its access, then S and T may perform conflicting accesses without being detected. OCTET thus requires a roundtrip communication (and the attendant happens-before relationship) between conflicting threads before allowing the conflicting thread to proceed. The protocol for handling conflicting accesses is shown in Figure 3.

[1] OCTET states are maintained at object granularity, so a conflicting access may not actually imply a dependence at the field level.

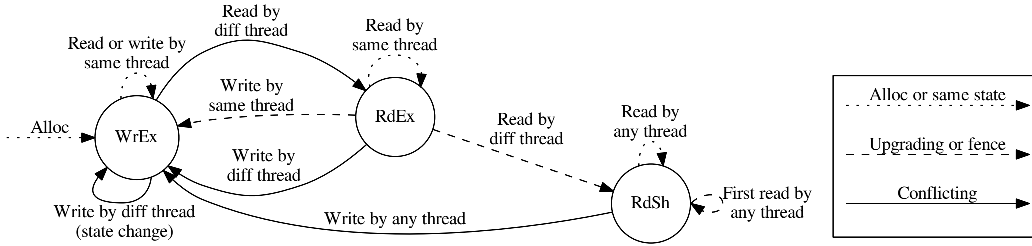

Figure 1. State diagram for OCTET. Before a program accesses an object, dynamic analysis checks that the object's state is compatible with the access (Figure 2). If so, the state stays the same (dotted self-loops). Otherwise, the state changes (upgrading and conflicting transitions). Note that to simplify the state diagram, we do not draw the Int state.

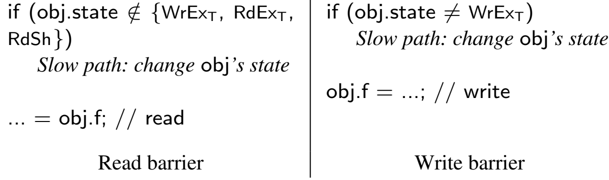

Figure 2. OCTET instrumentation at a program read and write. T is the current thread.

```
At safe points:
  currRequest ← request
  if (currRequest > response) // request seen
    memfence // ensure happens-before
    response ← currRequest
```

(a) Responding thread (thread A)

```
To move obj from WrEx_A or RdEx_A to WrEx_B:
  currState ← obj.state // WrEx_A or RdEx_A expected
  if (currState ≠ Int_* ∧ !CAS(obj.state, currState, Int_B))
    // obj.state ← Int_B failed
    check for and respond to requests, restart protocol;
  expectedResp ← ++A.request // atomic increment
  while (A.response < expectedResp)
    check for and respond to requests;
  memfence // ensure happens-before
  obj.state ← WrEx_B
  proceed with access
```

(b) Requesting thread (thread B)

Figure 3. Conflicting access protocol for conflicting accesses.

Consider two threads, A and B, where A, called the *responding* thread, has access to object obj in $\mathsf{WrEx_A}$ or $\mathsf{RdEx_A}$ state, and B, called the *requesting* thread, wants to write it. Every thread maintains two counters, a request counter and a response counter. The counters correspond to the number of requests for communication and the number of responses to communication requests, respectively. An invariant is maintained that request $\geq$ response. To gain access to obj, B places obj into an *intermediate* state, $\mathsf{Int_B}$, using compare-and-swap (CAS). This ensures that other threads attempting to access obj will wait until the conflict is resolved before proceeding, avoiding races in the protocol. Then, B notifies A of a request by atomically incrementing A's request counter, and saving its value locally.

Whenever A is at a *safe point* (a point known *not* to be in between a state check and an access), it examines its request and response counters. If request $>$ response, A knows at least one request was made to an object it has access to. At this point, A issues a fence and sets response $=$ request. B spins on A's response counter until it matches or exceeds its cached value, at which point it issues a fence, changes obj's state to $\mathsf{WrEx_B}$, and makes its access.

This protocol establishes two happens-before relationships: one between B's initial request for access to obj and any subsequent attempt by A to access obj (at which point A will find obj in $\mathsf{Int_B}$ state), and another between A's response to B and B's access to obj. The protocol is deadlock free: threads requests access to one object at a time, and while a requesting thread is awaiting a response it can respond to other requests itself. There is one complication: if a requesting thread finds an object in RdSh state, then *any* other thread may have read the object, which we address below.

2.2.2 Handling RdEx and RdSh states

If the only state in OCTET was WrEx, then *all* cross thread accesses to an object would appear to be conflicting accesses, and the roundtrip communication protocol outlined above would suffice to provide the necessary conflict detection and happens-before relationships. However, many shared objects are only or mostly read, not written, so treating every access as a conflicting access will result in significant unnecessary communication. OCTET uses the RdEx and RdSh states to mitigate this. The former state grants one thread read access to an object, while the latter allows *any* thread to read the object without synchronization.

If a thread A wants to write an object in $\mathsf{RdEx_A}$ state, it *upgrades* the state to $\mathsf{WrEx_A}$ with a CAS and proceeds (the CAS avoids races with threads moving the object to Int). If a thread A performs a conflicting read access to an object in WrEx state, it uses the roundtrip communication protocol to obtain the object in $\mathsf{RdEx_A}$ state. If A performs a conflicting *write* access to an object in RdSh state, OCTET requires that A perform roundtrip communication with *all* active threads, as any of them may have read the object.

To handle the remaining two transitions in Figure 1 (RdEx $\rightarrow$ RdSh and a thread reading a RdSh object for the first time), we introduce a *global* counter, gRdShCount, which tracks how many objects have been moved into RdSh state throughout the execution, and a local counter for each thread T, T.rdShCount.

When A wants to read an object that thread B has in $\mathsf{RdEx_B}$, A atomically increments gRdShCount and computes a RdSh state incorporating the new value of gRdShCount, CASing the object's state to this computed state. A then updates A.rdShCount with the new value of gRdShCount. Note that this protocol does *not* establish a happens-before relationship between A's and B's reads of the object. However, there is no dependence, so no relationship is necessary. The CAS *does* establish a happens-before relationship between B's changing the state to $\mathsf{RdEx_B}$ and A's read.

If a thread C wants to access an object in RdSh state, it extracts the counter value stored in the object's RdSh state, c, and checks if C.rdShCount $\leq c$. If so, C issues a fence and updates C.rdShCount to c before proceeding with its read. This ensures that C's read happens after the object was placed into RdSh state.

2.3 Correctness of OCTET

We now show that OCTET creates happens-before relationships between all cross-thread dependences. Note that OCTET does not concern itself with non-cross-thread dependences as they are enforced by the hardware and compiler.

We will assume, without loss of generality, that there is only one shared object, obj, and all cross-thread dependences arise through accesses to obj (interactions between multiple shared objects must happen within a single thread, and dependences between them will be preserved by the compiler's and hardware's reordering constraints). We also assume that OCTET's instrumentation behaves as expected, and hence OCTET ensures that an object is in a valid state before a thread performs its access (*e.g.*, for thread A to write obj, the object must be in state $\mathsf{WrEx_A}$).

Notation We denote a read by thread A as r_A, and a write by A as w_A. A dependence between two accesses is denoted with $\rightarrow$. Hence, flow (true) dependences are written $w \rightarrow r$, anti-dependences, $r \rightarrow w$, and output dependences, $w \rightarrow w$. A *cross-thread* dependence is a dependence whose source access is on one thread and whose dependent access is on another.

We will also use special notation for certain actions performed by threads when interacting with OCTET. $\mathsf{S}{\downarrow_\mathsf{A}}$ means that thread A put an object into OCTET state S. $\mathsf{resp_A}$ means that thread A issued an OCTET response (*i.e.*, updated its response counter).

Lemma 1. OCTET *creates a happens-before relationship to establish the order of every cross-thread dependence.*

Proof. We need only concern ourselves with cross-thread dependences that are not transitively implied by other dependences (cross thread or otherwise). We thus break the proof into several cases:

$w_A \rightarrow w_B$**:** OCTET's barriers enforce that when A writes obj, the object must be in $\mathsf{WrEx_A}$ state. When B attempts to perform its write, it will still find obj in $\mathsf{WrEx_A}$ (because the dependence is not transitively implied, no other conflicting access to obj could have happened in the interim). B will put obj into $\mathsf{Int_B}$ make a request to A. When A receives the request, it establishes $\mathsf{Int_B}{\downarrow_\mathsf{B}} \rightarrow_{hb} \mathsf{resp_A}$ and ensures that A will now see obj in state $\mathsf{Int_B}$ (preventing future reads and writes by A to obj). When B sees the update of A's response counter, it issues a fence, moves obj to state $\mathsf{WrEx_B}$ and proceeds with its write, establishing $w_A \rightarrow_{hb} w_B$.

$r_A \rightarrow w_B$**:** There are two cases to deal with for this dependence.

Case 1: B finds the object in an exclusive state (either $\mathsf{RdEx_A}$ or $\mathsf{WrEx_A}$). $r_A \rightarrow_{hb} w_B$ is established by the same roundtrip mechanism as in the prior scenario.

Case 2: B finds the object in RdSh state. In this case, the protocol for dealing with RdSh objects, described above, requires that B perform a roundtrip communication with *all* threads, establishing $r_A \rightarrow_{hb} w_B$.

$w_A \rightarrow r_B$**:** For thread A to write to obj, the object must be in $\mathsf{WrEx_A}$ state. Then there are three scenarios by which this dependence could occur.

Case 1: B is the first thread to read obj after the write by A, so it will find obj in $\mathsf{WrEx_A}$ state. This triggers roundtrip communication and establishes $w_A \rightarrow_{hb} r_B$.

Case 2: B is the second thread to read obj after the write by A. This means that there was some thread, C, that left the object in state $\mathsf{RdEx_C}$. By the previous case, we know $w_A \rightarrow_{hb} \mathsf{RdEx_C}{\downarrow_\mathsf{C}}$, with a fence between $\mathsf{resp_A}$ and $\mathsf{RdEx_C}\downarrow_\mathsf{C}$. Hence, when B uses a CAS to move the object to state RdSh, it establishes $\mathsf{resp_A} \rightarrow_{hb} \mathsf{RdSh}{\downarrow_\mathsf{B}}$, enforcing $w_A \rightarrow_{hb} r_B$ transitively.

Case 3: B finds obj in RdSh state upon reading it. Note that by the previous case, there must be some thread C that placed obj in RdSh (establishing $w_A \rightarrow_{hb} \mathsf{RdSh}{\downarrow_\mathsf{C}}$). To access obj in RdSh state, B accesses B.rdShCount and the counter value stored as part of the obj's state as described in Section 2.2.2, ensuring that B last saw the value of gRdShCount *no earlier* than when C put obj in RdSh. Hence, we have $\mathsf{RdSh}{\downarrow_\mathsf{C}} \rightarrow_{hb} r_B$, establishing $w_A \rightarrow_{hb} r_B$ transitively.

Thus, OCTET establishes a happens-before relationship between the accesses of every cross-thread dependence. ☐

Note that by synchronizing on all conflicting shared-memory accesses, OCTET provides sequential consistency [26] with respect to the *compiled* program, even on weak hardware memory models [1].

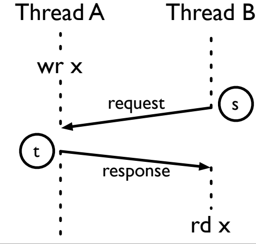

Figure 4. A simple program illustrating the behavior of record and replay using OCTET

3. Record & Replay

This section presents ROCTET, a system for record and replay built on top of OCTET. The basic idea behind the approach is to *record* the timing of all cross-thread dependences by noting where in their respective executions two threads are when OCTET detects cross-thread interactions. We can then *replay* the execution by manipulating threads' execution so that all the dependences logged during the record phase occur in the same order during replay.

3.1 A simple example

To understand the intuition behind our approach to record and replay, consider the simple program shown in Figure 4, with two threads, A and B. Among other operations, A and B have one cross-thread dependence through object obj: A writes to obj and later B reads from it. Since obj is being tracked by OCTET, it must be in state $\mathsf{WrEx_A}$ at A's write. When B wants to read obj, it sends a request to A at time s (the solid "request" line) and A responds at time t (the solid "response" line), allowing B to move obj into $\mathsf{RdEx_B}$. OCTET's protocols ensures that a happens-before relation exists between A's write and B's read.

To correctly replay this behavior, ROCTET records the time s at which B made its request and the time t at which A responded. On replay, when B reaches s, B pauses its execution until A has passed point t, guaranteeing that A has already performed the write to obj. We can then let B proceed, reading obj and observing the proper value.

This approach generalizes to multiple threads accessing multiple shared objects: ROCTET records the request and response times of all communicating threads during the record phase, and during replay delays requesting threads until responding threads pass the appropriate response point before continuing.

We now answer two key questions. First, how does ROCTET determine thread timings during record runs (*i.e.*, what information makes up the s and t time points from Figure 4)? Second, how does ROCTET ensure that dependences are preserved during replay?

3.2 Recording execution

During recording, each thread needs to keep track of the communication it has performed. For any (round-trip) communication, a thread can either be a *requesting* thread (it needs access to an object that other threads have read/written last) or a *responding* thread (it has access to an object, and it must release that object to another thread). Following the lead of Figure 4, we will assume the requesting thread is named B and the responding thread, A. Let us consider the behaviors separately:

The requesting thread: If thread B requests an object from another thread, A, it must record two pieces of information in a *request log*: where it is in its execution (*i.e.*, the dynamic program location) and where A is in its execution. The first piece of information can be tracked in many ways. For example, we could track the program counter (static program location) as well as a thread-local counter that tracks how many loop back edges and function calls have been encountered. Section 4 discusses our approach to tracking dynamic program location.

Tracking where A is requires more care. A's dynamic program counter is thread local and updated without synchronization, so racy reads of that information may produce spurious values. Instead, we note that OCTET's communication protocol requires that B calculate A's expected response counter, which reflects the total number of responses A will have made (including this one) by the time it responds to B, and captures sufficient information about A's location. Hence, in addition to its dynamic program count, B stores A's identity and its current request counter (which is a lower bound on the eventual value of A's response counter) in the request log.

The responding thread: If thread A *responds* to another thread, it records its dynamic program counter in a *response log*, capturing where it is when it responds. In addition, because requesting threads will need to know where A was when it responded, A records its current response counter in the response log. A's logging of its response counter is the counterpart to the requesting thread's logging of A's request counter, and allows the requests and responses to be aligned.

The above protocol addresses threads' behavior during state transitions that trigger roundtrip communication However, there are two state transitions that imply interaction between threads but do *not* perform roundtrip communication: transitions to RdSh and accesses to objects that are already in RdSh. What information must be recorded to ensure the order of these accesses is captured?

Recall that if thread B moves an object obj from RdEx_A to RdSh, it updates the global read-shared counter, gRdShCount, and performs a simple compare-and-swap on the object state. By recording the update to gRdShCount in the log, we can ensure that this transition to RdSh happens according to a fixed total order with all other transitions to RdSh. However, we must also track where thread A is. Logging A's response counter is insufficient, as its value may be from *before* A read obj (and hence may not ensure that A is past its read). To address this problem, we introduce a new thread-local counter, RdExTrans_A, that keeps track of how many times a thread A has moved an object into RdEx state. Thus, when A moves obj into RdEx_A, it atomically updates its value of RdExTrans_A. When moving obj to RdSh, B logs A's value of RdExTrans_A as its estimate of where A is.

When a thread reads from an object obj that is *already* in RdSh state, if OCTET requires an update of the thread's local read-shared counter, .rdShCount, then that counter value is recorded in the request log. This data provides enough information to ensure that this read happens after obj was moved to RdSh state. If the read-shared counter was not updated, then other entries in the log will already capture that this read happens after obj was moved to RdSh.

3.3 Replaying execution deterministically

To replay an execution recorded using the above strategy, we must make sure that the threads' execution interleaves such that the order of each event we recorded in the logs is preserved. Interestingly, we do *not* need to replay the order of synchronization events (lock acquires and releases, waits, notifies, etc.); merely respecting the original dependences of the program suffices. Indeed, replaying high-level synchronization may result in deadlock as high level races (acquiring locks in a different order) may preclude achieving the proper interleaving of memory accesses within critical sections. Hence, we elide all synchronization events from the replayed run. We also no longer need to track cross-thread interactions, as we are replaying them, so OCTET is turned off, and object state is no longer maintained. The only information we need concerns the progress of the threads: we track each thread's dynamic program counter, response counter and RdExTrans counter as well as the global read-shared counter.

Replaying a thread's execution involves reading back the thread's request and response logs to ensure that its progress lines up correctly with other threads' progress. As a thread executes, it checks its dynamic program counter and performs an action whenever an entry from its logs matches the program counter.

Consider the behavior of a particular thread, A. An entry in A's response log where it changed its response counter indicates that A responded to a communication request from another thread. A merely executes a memory fence and updates its response counter as specified. The fence ensures that any thread reading A's response counter can use it as an accurate gauge of A's progress. A similarly updates its value of RdExTrans_A as specified by the response log.

An entry in A's request log indicates that A is about to access an object and needs to coordinate with one or more other threads. If the log entry contains the ID of a second thread, B, and the value of B's request counter, A pauses until B's response counter matches or exceeds the specified value. A then executes a fence and proceeds, correctly performing its access *after* B performed its prior conflicting access. If the entry specifies that gRdShCount should be changed (because, at this point during recording, A moved an object from RdEx to RdSh), A waits until gRdShCount reaches the appropriate prior value (indicating that all prior RdSh changes have occurred), then atomically updates gRdShCount before proceeding with its read. Finally, if the entry is simply a value of gRdShCount (because of a fenced read to a RdSh object during recording), A waits until gRdShCount reaches or exceeds the specified value, issues a fence, and continues.

While the specific manipulations performed by the threads' processing their request logs seem complicated, we note that their entire purpose is to ensure that a happens-before relationship is established between one thread's access to an object and any later thread's dependent access to that same object. Moreover, this happens-before relationship enforces exactly the same order of conflicting accesses as occurred in the original recorded execution, providing deterministic replay.

Because replay performs essentially the same operations as record, the primary overhead of replay is threads' waiting at communication points. Hence, we expect replay to have similar performance and scalability as record.

3.4 Soundness of Record & Replay

ROCTET's correctness relies on the observation that value determinism (all reads performed by a replayed program produce the same results as the original recorded program) 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. As before, we assume without loss of generality that there is a single shared object obj, and all cross-thread dependences arise through accesses to obj.

We have already shown that OCTET creates happens-before relationships between all cross-thread dependences in the recorded run (Lemma 1). We now show that the information ROCTET logs during a recorded run is sufficient to allow the replayed run to deterministically replay the recorded execution.

Theorem 1. *Given logs recorded during an execution,* ROCTET *is able to deterministically replay that execution.*

Proof. We proceed by showing that every cross-thread dependence in the recorded run (hereafter referred to as rec) is respected by the replayed run (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.

$w_A \rightarrow w_B$ In rec, this write dependence is captured by OCTET as a transition from $\mathsf{WrEx_A}$ to $\mathsf{WrEx_B}$, with B as the requesting thread and A as the responding thread. A's response log notes the recorded value of its response counter, rc_a, with a precise count of how many responses A had made prior to this point, as well as its dynamic program location, while B's request log notes the expected value of A's response counter, rc_b, with $rc_b > rc_a$. As A runs rep, it maintains a response counter, rc. When A reaches the dynamic program point where the communication occurred, $rc = rc_a$. It then increments rc by the number of responses it made at this point, so $rc \geq rc_b$. When B reaches the point where it made the request, it compares rc_b to rc. If $rc \geq rc_b$, B can be sure that A has already performed its write, and hence B's write will happen later, preserving the dependence.

$r_A \rightarrow w_B$ In rec, prior to performing w_B, B will find obj in either state $\mathsf{RdEx_A}$ or RdSh. In either case, OCTET initiates roundtrip communication between A and B. This scenario is therefore analogous to that of the previous case, and w_B will occur after r_A in rep, preserving the dependence.

$w_A \rightarrow r_B$ There are three possible types of OCTET state transitions in rec that might arise due to this dependence. (i) $\mathsf{WrEx_A} \rightarrow \mathsf{RdEx_B}$ requires roundtrip communication, and would be enforced as in the previous cases. (ii) If $\mathsf{WrEx_A} \rightarrow \mathsf{RdEx_C} \rightarrow$ RdSh, B reads obj after some third thread put it into $\mathsf{RdEx_C}$. In this case, replay uses a similar mechanism as above to ensure that C has moved past the point where it put obj into $\mathsf{RdEx_C}$, but using the $\mathsf{RdExTrans_C}$ counter instead of the response counter. (iii) r_B could happen when obj was already in RdSh state. In this case, we note that updates to gRdShCount are replayed at the correct times, and that B would record, in its request log, what the value of gRdShCount was before it performed r_B in rec. Before performing r_B in rep, B ensures that gRdShCount has at least the recorded value. This ensures that *all* RdEx $\rightarrow$ RdSh transitions that happened before r_B 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. □

4. Implementation

We have implemented OCTET and ROCTET in Jikes RVM, a high-performance Java-in-Java virtual machine [2, 3].[2]

[2] `http://www.jikesrvm.org`

4.1 OCTET instrumentation and metadata

Adding* OCTET*'s instrumentation. Jikes RVM uses two dynamic compilers to transform bytecode into native code. The *baseline* compiler compiles each method when it first executes, converting bytecode directly to native code. The *optimizing* compiler recompiles hot (frequently executed) methods at increasing levels of optimization [5]. We modify both compilers in order to add instrumentation (*cf.* Figure 2) at every operation that reads or writes an object field, array element, or static field.

The implementation adds instrumentation to application methods and Java library methods (*e.g.*, java.*). A significant fraction of OCTET's overhead comes from instrumentation in the libraries, which must be instrumented in order to properly replay conflicting dependences that occur within them.

OCTET *metadata.* In order to store OCTET state, the implementation adds one metadata word per (scalar or array) object by adding a word to the header, and one word per static field by inserting an extra word per field into the global table of statics. The implementation represents $\mathsf{WrEx_T}$ and $\mathsf{RdEx_T}$ by storing the address of a thread object T, using one low bit to distinguish $\mathsf{WrEx_T}$ from $\mathsf{RdEx_T}$. Jikes RVM reserves a register that always holds the current thread object, so checking whether a state is WrEx or RdEx is extremely fast. To represent RdSh, the implementation simply uses the object's RdSh counter, using a number range that cannot overlap with thread addresses. To check whether a state is RdSh *and* the thread's .rdShCount counter is up-to-date with the object's RdSh counter, the instrumentation compares the metadata word with .rdShCount.

The implementation initializes the OCTET state of newly allocated objects and newly resolved static fields to the $\mathsf{WrEx_T}$ state, where T is the allocating/resolving thread. Since the Java Memory Model does not provide a happens-before edge between object allocation and another thread's use of the object [30], OCTET instrumentation might see an uninitialized metadata word (guaranteed to be zero), which will trigger the instrumentation slow path, which will wait for the value to become initialized.

Optimization potential. OCTET adds instrumentation at nearly every access, except those to objects and fields that Jikes RVM's escape analysis determines are thread local and thus cannot be involved in a conflicting access. While this simple optimization has a modest effect on performance (improving it by 5% on average), more advanced escape analyses could be more aggressive at excluding accesses from instrumentation. Furthermore, if the code accesses the same object or static field twice without an intervening safe point, the compiler could elide instrumentation from the second access, as it cannot trigger a conflicting access.

To some extent, OCTET obviates the need for static optimizations to identify accesses that cannot conflict, because it adds such low overhead at non-conflicting accesses. Nonetheless, such optimizations would help lower OCTET's basic instrumentation overhead (*cf.* the *Octet w/o comm* bar in Figure 5.2).

4.2 OCTET conflicting transitions

Receiving requests at safe points. Threads respond to requests only when they reach safe points, which must not be between the state check and use for an access (Section 2.2.1). Safe points include *yield* and *block* points. Yield points are already inserted into program code by most or all managed-language VMs. Each yield point checks a thread-local flag indicating whether the thread should stop, *e.g.*, for garbage collection (GC) and profiling. When a requesting thread increments a responding thread's request counter, the requesting thread also sets the responding thread's yield point flag.

Block points occur when a thread enters VM code and might block for a while (*e.g.*, waiting to acquire a monitor or waiting

for other threads to join a GC), or when a thread is waiting in OCTET for a communication response as part of a conflicting state change. We have identified block points in Jikes RVM and modified these points so they (atomically) set a thread-local "block" flag while the thread is blocked. If a requesting thread T2 observes (atomically) a responding thread T1 at a block point, then T1 has *implicitly* responded, and T2 can proceed immediately to change the object's state. Avoiding explicit communication with threads at block points is especially useful when threads outnumber cores, since unscheduled threads are often at block points.

Waiting for responses. Unless a requesting thread finds its communicating partner in the blocked state (and can observe an implicit response, as described above), the requesting thread must wait for the responding thread to respond. As most responses occur quickly, but some take much longer, the requesting thread first uses spin-waiting, then switches to thread yielding, and finally waits on a pthread monitor. A responding thread always broadcasts on the requesting thread's monitor, in case the requesting thread is waiting.

4.3 Implementing ROCTET

This section describes our implementation of ROCTET for providing high-performance record and replay. Note that due to the difficulty of controlling various sources of nondeterminism, we do not demonstrate replay of the high-performance execution (instead, Section 5.3 demonstrates record and replay in a more controlled environment). Here we describe changes specific to high-performance recording, which also apply to high-performance replay (though we do not evaluate the latter).

Our ROCTET implementation makes two main changes on top of OCTET. First, it tracks the dynamic program location by incrementing a thread-local counter at every yield point (method entries and loop back edges). Second, the implementation records and replays events needed to enforce the same happens-before relationship in both exections. It records conflicting transition requests and responses, as well as RdEx→RdSh transitions and RdSh→RdSh transitions that need fences, in *request* and *response* logs, as described in Section 3. Each log entry includes the value of dynamic program location counter, as well as information specific to the type of event being recorded.

During replay, the compiler removes most synchronization operations, as their effects are provided by replay's behavior. The compiler removes lock acquires and releases (corresponding to Java synchronized blocks) from application and library methods. When the program executes a wait or notify operation without holding the corresponding lock, it ignores the operation. Fork–join synchronization is preserved.

5. Evaluation

Experimental setup. To account for run-to-run variability due to dynamic optimization guided by timer-based sampling, we execute 15 trials for each performance result and take the median. We build a high-performance configuration of Jikes RVM (FastAdaptive) that optimizes the VM and adaptively optimizes the application as it runs. We use Jikes RVM's high-performance generational Immix collector [11] and let the VM choose its own heap size adaptively.

Benchmarks. In our experiments, our modified Jikes RVM executes the parallel DaCapo Benchmarks [10] (version 2006-MR2) and a fixed-workload version of SPEC JBB2000 called pseudojbb [45]. We omit the parallel DaCapo benchmark lusearch because we could not get it to execute correctly with our changes.

Platform. Our experiments execute on a 4-core Intel i5 3.2-GHz system with 4 GB main memory running 64-bit Linux 2.6.32.

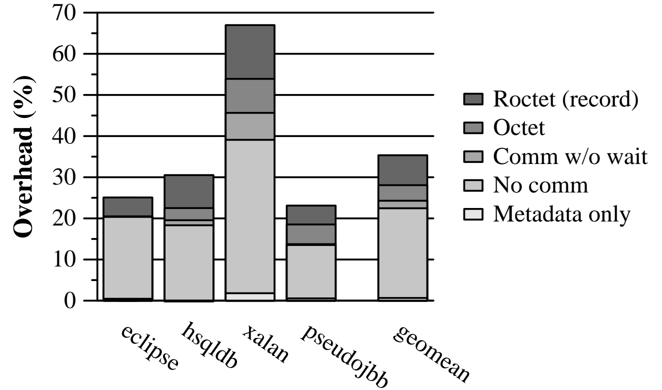

Figure 5. Performance of OCTET and ROCTET.

5.1 OCTET state transitions

Table 1 shows the number of transitions recorded. We use a profiling configuration of OCTET that adds instrumentation to count these transitions. The three groups of columns show increasing levels of synchronization that correspond with the transitions shown in Figure 1. *Alloc or same state* transitions (dotted lines in Figure 1) use no synchronization. *Upgrading or fence* transitions (dashed lines in the figure) require a compare-and-swap or a fence operation. *Conflicting* transitions (solid lines in the figure) require the request-response protocol.

As seen in the table, the vast majority of accesses do not require synchronization. Lightweight fast-path instrumentation handles these transitions (Section 2). *Upgrading and fence* and *Conflicting* transitions occur in similar numbers; the conflicting transitions are more of a concern because we expect them to be significantly more expensive. Conflicting transitions range from fewer than 0.001% (eclipse) to 0.15% (xalan) of all transitions. These results provide evidence in support of OCTET's optimistic approach.

5.2 Performance

Time overhead. Figure 5.2 presents the runtime overhead of OCTET and ROCTET. The overheads are normalized to unmodified Jikes RVM. Each bar represents the overhead of ROCTET, with sub-bars representing subsets of functionality. *Metadata only* presents the runtime overhead of adding a word to each object's header (and static field) and initializing it to WrEx upon allocation, and is less than 1% on average.

The configuration *No comm* adds OCTET barriers, but does not perform the conflicting transition protocol, measuring only OCTET's instrumentation overhead. (We still allow state transitions to occur in this configuration; disabling them actually *slows* execution, as many fast-path checks fail since objects are often in a conflicting state.) OCTET's instrumentation adds 22% overhead on average, and almost 40% to xalan, which has a high density of reads and writes. While this overhead is likely acceptable for many production environments, there is significant potential for improving it with known compiler optimmizations (Section 4.1).

Comm w/o wait performs the conflicting transition protocol, except requesting threads do not actually wait to receive responses before continuing execution, and adds 2% average overhead. When requesting threads correctly wait for responses (the *Octet* configuration), overhead increases by an additional 4% on average. Unsurprisingly, xalan, which experiences an above-average fraction of conflicting transitions (Table 1) adds more communication overhead than average; and eclipse, which experiences relatively few conflicting transitions, adds less. Overall OCTET overhead is 28% on average.

	Alloc or same state				Upgrading or fence			Conflicting			
	Alloc	WrEx	RdEx	RdSh	RdEx→WrEx	RdEx→RdSh	RdSh	WrEx→WrEx	WrEx→RdEx	RdEx→WrEx	RdSh→WrEx
eclipse	14,935,465,758 (99.9985%)				93,553 (0.00063%)			126,917 (0.00085%)			
	2.2%	87%	2.9%	8.0%	0.000090%	0.00034%	0.00019%	0.000017%	0.00073%	0.000014%	0.000092%
hsqldb	719,986,050 (99.85%)				486,305 (0.067%)			588,114 (0.082%)			
	3.5%	91.2%	0.41%	4.8%	0.033%	0.015%	0.020%	0.020%	0.054%	0.00053%	0.0070%
xalan	9,661,511,012 (99.74%)				9,884,204 (0.10%)			14,937,309 (0.15%)			
	1.3%	83%	0.27%	15%	0.10%	0.000029%	0.000070%	0.052%	0.10%	0.0000071%	0.0000067%
pseudojbb	1,846,344,850 (99.90%)				938,914 (0.051%)			936,637 (0.051%)			
	3.5%	81%	1.6%	14%	0.043%	0.0050%	0.0029%	0.00050%	0.050%	0.0000024%	0.00029%

Table 1. OCTET object and field state transitions, including fast-path executions that do not change the state. The first row for each benchmark is the transitions for each column group and percentage of all transitions, and the second row shows transitions of each type as a percentage of all transitions. We round percentages x as much as possible such that x and $100\% - x$ each have at least two significant digits.

	Threads		Sum of log sizes	
	Total	Max live	Requesting	Responding
eclipse	16	8	4 MB	<1 MB
hsqldb	402	102	19 MB	3 MB
xalan	9	9	204 MB	92 MB
pseudojbb	37	9	14 MB	2 MB

Table 2. The total number of threads executed by each program and the maximum number that are running at any time (Columns 2 and 3). The sum across threads of the ROCTET requesting and responding log sizes (Columns 4 and 5).

The *Roctet (record)* configuration adds instrumentation to record program execution as described in Section 4.3. On average, *Roctet (record)* adds 7% time over *Octet*—35% over program execution. Much of ROCTET's overhead, especially for xalan, is due to the file I/O involved. ROCTET currently uses synchronous I/O to maintain the logs, but a more sophisticated, asynchronous solution could perform significantly better.

Table 2 shows the number of application threads that each program executes: the total number of threads executed (Column 1) and the maximum number of threads that are running at any time (Column 2). It also reports the total size of all threads' logs for conflicting transition requests and RdSh-related transitions (Columns 3) and responses to conflicting transition requests (Column 4). Log size is well correlated with number of conflicting transitions and RdSh-related transitions that create a log entry. We note that our log entries are not optimally compact, e.g., each entry could save bits by storing deltas from the last entry's values, instead of storing full values.

We note that our prototype implementation cannot currently replay these recorded executions faithfully, as explained next.

5.3 Testing Record & Replay

ROCTET records enough information about threads' accesses to shared memory to control the nondeterminism that arises through cross-thread dependences. Unfortunately, cross-thread dependences are not the only source of nondeterminism in a program's execution. Jikes RVM has many difficult-to-control sources of nondeterminism, such as timer-based sampling and synchronization via low-level atomic operations (e.g., compare-and-swap). Parallel garbage collection (GC) threads trace and nondeterministically reorganize the heap. Different heap layouts lead to different application behavior (*e.g.*, different Object.hashcode() values affect the iteration order of data structures such as HashMap). Other sources of nondeterminism include I/O in the VM and libraries, and, most perniciously, timer-based sampling that leads to different JIT compilation decisions.

Prior work has largely been able to address these challenges for two main reasons. First, most prior work records and replays native (C/C++) programs, instead of managed-language programs running in a VM. Second, most work sidesteps the challenge of recording all sources of multithreaded nondeterminism. For example, Respec does not identify nondeterminism resulting from the behavior of custom synchronization primitives, but instead relies on checkpointing and rollback if custom synchronization results in nondeterminism [28].

Methodology. Our implementation supports a "replayable" configuration that aims to provide limited record and replay, despite the myriad sources of nondeterminism. The changes made to the replayable configuration can be split into two categories. Those sources of nondeterminism we control (providing deterministic replay with respect to their effects) and those we ignore (allowing nondeterministic replay with respect to their effects).

Because JIT compilation in Jikes RVM is nondeterministic, our replayable configuration uses the baseline compiler. While this means that overhead estimates of the replayable configuration are not meaningful, we can still test ROCTET's control of cross-thread shared memory accesses. We limit nondeterminism due to garbage collection by (i) using a single-threaded GC, and (ii) recording GC points and performing GC at exactly those points during replay. Nondeterminism through java.util.Random is eliminated by using a fixed (rather than timer-based) seed.

We ignore nondeterminism arising from iteration over unordered data structures such as java.util.HashMap, whose behavior is dependent on heap layout, by not instrumenting those operations. We similarly ignore nondeterminism arising from I/O operations (while I/O could be recorded and replayed by a separate process, we do not address that in this paper). Class loading also presents difficulties, as the thread that loads a class is chosen nondeterministically. While a thread performs class loading, a thread local flag is set that precludes ROCTET from recording execution.

When nondeterministic code executes, it can perturb the value of the dynamic program counter, making the counter useless during replay. The replayable configuration thus uses *per-site* counters. Each (static) site increments its own counter, so nondeterminism at one site will not necessarily affect another site's counter. The implementation maintains a counter for every read and write. Each recorded event includes its site and its (per-thread) per-site counter.

Results. Using the methodology described above, we have been able to replay hsqldb and xalan successfully. We have not been able to replay eclipse and pseudojbb, as we have not found all sources of non-determinism to either control or ignore.

For hsqldb and xalan, we found that replay successfully executes each program based on per-thread logs. Note that each request and response must be replayed exactly. If requests and responses do not match up, replay will throw an error, while if nondeterminism occurs, the site counters may not be accurate and the program may deadlock. Further, all tracked per-site counters had matching values after both record and replay, indicating that the program executed deterministically.

Nevertheless, without logging the value of every read performed by hsqldb and xalan, we cannot be sure that replay is truly deterministic. We may have "gotten lucky," and replay matched record even while ROCTET mistakenly allowed nondeterminism.

To gain additional confidence that deterministic replay was usefully controlling the nondeterminism of thread accesses, we performed two experiments. In the first, we execute replay but *enable* synchronization. We expect this to cause deadlocks: if a high level race (*e.g.*, a race in the order of lock acquisition) occurs, then the "wrong" thread might acquire a lock first, preventing the right thread from entering a critical section and responding to requests for an object. Indeed, we find exactly this behavior: both programs deadlock during replay with synchronization. This result suggests that these programs have high-level nondeterminism that is suppressed by ROCTET.

The second experiment checks whether the programs will execute correctly both without synchronization and without ROCTET's replay controlling the interleavings. If these programs cannot execute correctly without synchronization, that indicates replay is doing *something* useful by controlling the ordering of reads and writes, since replayed execution without synchronization executes correctly. We find that both programs fail when executed without synchronization and without replay, suggesting that replay is at least replaying reads and writes in an order that respects the program's mutual exclusion and high-level ordering requirements.

These experiments demonstrate our record and replay system to a limited extent, and they provide some confidence in its claims of record and replay. However, we have not presented a practical system for record and replay. The main contribution of this paper is a new approach for tracking conflicting accesses efficiently, which we demonstrate can be applied to recording conflicting accesses, in order to replay them later.

6. Related Work

6.1 Record and replay

Record and replay of multithreaded programs may support offline or online replay, or both. *Offline* replay allows programmers to reproduce production failures that may be difficult to reproduce otherwise due to nondeterminism. *Online* replay allows multiple machines to execute the same interleavings at the same time, enabling fault tolerance [13] and offloading of dynamic analysis [15, 28].

Record and replay for uniprocessors is relatively straightforward: It is sufficient to record context switches and nondeterministic system events such as I/O and reading the system clock. Record and replay on multiprocessor systems is harder due to nondeterministic interleavings between threads. These interleavings occur due to (1) high-level races between synchronization variables (e.g., lock acquires and releases, wait-notify, fork-join, Java volatile reads and writes) and (2) data races between unsynchronized reads and writes. Prior work that records all synchronization operations can be efficient because synchronization operations are relatively infrequent, but this approach does not guarantee replay for racy executions [41]. Recording data races adds high overhead because every read and write is a potential data race, and synchronized analysis is needed to capture dependences correctly [27].

Recent solutions for multiprocessor record and replay do not record conflicting dependences explicitly. *Respec* supports *online* replay by recording synchronization operations and speculating that most data races do not lead to external effects [28]. It rolls back to a prior checkpoint on a misspeculation. It cannot provide *offline* replay without additional support such as probabilistic search. Other approaches offer probabilistic *offline* replay based on reproducing executions from limited recorded information [4, 38, 50], but do not support online replay, and are not guaranteed to reproduce an execution within a bounded number of attempts.

DoublePlay supports both online and offline replay using *uniparallelism* to execute overlapping serial program intervals using hints from a parallel execution [46]. It records checkpoints in order to roll back in the event of a misspeculation. DoublePlay needs double the number of cores as the original program in order to provide low overhead. It is unclear whether this overhead, which is 15% for two worker threads and 28% for four threads, will scale well with additional threads. Without extra cores, DoublePlay adds 100% overhead.

Hardware support. Custom hardware support can achieve low overhead record and replay by piggybacking on the hardware cache coherence protocol [21, 22, 31, 33, 39, 51]. However, manufacturers have been slow to adopt such hardware support, which would add complexity to already-complex coherence protocols, and programmers and users are not clamoring for such support. We believe that manufacturers will add such hardware support if demand for it grows. ROCTET has the potential to make software-based record and replay fast enough for widespread use in production systems, ultimately leading programmers and users to demand—and thus probably get—even faster support in hardware.

Dunlap et al. achieve record and replay in commodity hardware by using virtual memory page protection to trigger hardware traps at potentially conflicting accesses [19]. Because this approach uses page granularity, it adds high overhead due to false sharing on some applications.

Static analysis. Static program analysis can identify reads and writes that cannot be part of a data race [32, 49]. However, prior work that applies static race detection to dynamic race detection still add significant overhead because conservative static analysis still identifies many accesses as potential races [14, 48].

Determinism. An alternative to record and replay is for the runtime system to execute multithreaded programs deterministically [8, 9, 16–18, 34, 35]. Runtime determinism approaches face similar performance challenges as record and replay. They either do not handle racy programs [34, 35], add high overhead [8, 16], handle only fork-join parallelism efficiently [9], or require custom hardware [17, 18].

Languages such as Deterministic Parallel Java and Jade provide determinism at the language level [12, 40], while Determinator provides programming model and operating system support to guarantee determinism [6]. Programmers must rewrite their programs to use these approaches.

6.2 Tracking ownership

Biased locking. Prior work proposes *biased locking* as an optimistic mechanism for performing lock acquires without atomic operations [23, 36, 42], now implemented in major commercial Java virtual machines. Each lock is "biased" toward a particular thread that can acquire the lock without synchronization; other threads must communicate with the biasing thread before acquiring the lock. In contrast, OCTET applies an optimistic approach to all program accesses, not just locks. OCTET introduces WrEx, RdEx, and RdSh states in order to support different sharing patterns efficiently. OCTET's conflicting transition protocol is lightweight compared to biased locking's communication mechanisms, which is important since regular memory accesses occur more frequently than synchronization operations.

Hindman and Grossman present an approach similar to biased locking for tracking reads and writes in software transactional memory [20]. Their approach is not as lightweight as OCTET's, so it would be difficult to apply to all reads and writes efficiently,

and it will not handle read-shared access patterns efficiently since it does not introduce states as OCTET does.

Cache coherence. OCTET's states correspond to cache coherence protocol states; its conflicting state transitions correspond to remote invalidations (Section 2.1). Thus, program behavior that leads to expensive OCTET behavior already has poor behavior due to remote cache misses.

Cache coherence has been implemented in software in distributed shared memory (DSM) systems to reduce coherence traffic [7, 24, 25, 29, 43, 44]. *Shasta* and *Blizzard-S* both tag shared memory blocks with coherence states, which instrumentation at each access checks [43, 44]. A coherence miss triggers a software coherence requests; processors periodically poll for such requests.

While each unit of shared memory can have different states in different caches in cache coherence approaches, each unit of shared memory has one state in OCTET. While cache coherence provides data consistency, OCTET provides concurrency control for dynamic analyses that check or enforce correctness properties, such as record & replay.

Object tracking. Von Praun and Gross describe an approach for detecting races in shared-memory programs based on tracking the ownership of objects [47]. Their ownership system is used to dynamically identify shared objects, allowing their race detector to restrict its attention to those shared objects. At a high level, their approach tracks object states and ownership by threads in a manner similar to OCTET. While ROCTET could, in principle, be built on von Praun and Gross's ownership model rather than OCTET, there are some drawbacks. Their ownership model allows objects in an exclusive state to avoid synchronization, but objects in a shared–modified or shared–read state require synchronization on every access; objects that enter shared states cannot return to an exclusive state. In contrast, OCTET supports transitioning back to exclusive states, and object accesses require synchronization only on state changes. OCTET's more precise tracking of ownership thus requires less synchronization when used for record & replay.

7. Conclusion

Efficient, software-based record and replay is a useful tool in a developer's toolkit. Unfortunately, most software record and replay approaches incur high overheads due to the cost of accounting for shared-memory interactions between threads. We presented a novel dynamic analysis, OCTET, which tracks object states throughout execution, and can detect cross-thread interactions at an overhead proportional to the number of conflicting accesses, rather than all accesses. We demonstrated the utility of OCTET by building a prototype record-and-replay system, ROCTET, on top of it. On four benchmarks, we found that ROCTET can record sufficient information to deterministically replay execution with an average overhead of 35%. This compares very favorably with the overheads of prior general-purpose record and replay systems (the best of which has overheads of 100%, albeit on a different set of benchmarks). We also showed that ROCTET's logs contain sufficient information to successfully replay two applications in a controlled environment. We have thus demonstrated that general, software-only, record can be achieved at low overhead.

Acknowledgments

Thanks to Swarnendu Biswas, Meisam Fathi Salmi, Jipeng Huang, Aritra Sengupta, and Minjia Zhang for helpful feedback on the paper and implementation. Thanks to Brian Demsky for helpful discussions about memory models.

References

[1] S. V. Adve and H.-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *Communications of the ACM*, 53:90–101, 2010.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[3] 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.

[4] G. Altekar and I. Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *ACM Symposium on Operating Systems Principles*, pages 193–206, 2009.

[5] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, 2000.

[6] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient System-Enforced Deterministic Parallelism. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, 2010.

[7] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language For Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18:190–205, 1992.

[8] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–64, 2010.

[9] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 81–96, 2009.

[10] 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 *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, 2006.

[11] S. M. Blackburn and K. S. McKinley. Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *ACM Conference on Programming Language Design and Implementation*, pages 22–32, 2008.

[12] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel Programming Must Be Deterministic by Default. In *USENIX Conference on Hot Topics in Parallelism*, pages 4–9, 2009.

[13] T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault Tolerance. In *ACM Symposium on Operating Systems Principles*, pages 1–11, 1995.

[14] 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 *ACM Conference on Programming Language Design and Implementation*, pages 258–269, 2002.

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

[16] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable Deterministic Multithreading Through Schedule Memoization. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 1–13, 2010.

[17] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2009.

[18] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: A Relaxed Consistency Deterministic Computer. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 67–78, 2011.

[19] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution Replay of Multiprocessor Virtual Machines. In *ACM/USENIX International Conference on Virtual Execution Environments*, pages 121–130, 2008.

[20] B. Hindman and D. Grossman. Atomicity via Source-to-Source Translation. In *ACM SIGPLAN Workshop on Memory System Performance and Correctness*, pages 82–91, 2006.

[21] D. R. Hower and M. D. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *ACM/IEEE International Symposium on Computer Architecture*, pages 265–276, 2008.

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

[23] K. Kawachiya, A. Koseki, and T. Onodera. Lock Reservation: Java Locks Can Mostly Do Without Atomic Operations. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–141, 2002.

[24] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *USENIX Technical Conference*, pages 115–132, 1994.

[25] L. I. Kontothanassis and M. L. Scott. Software Cache Coherence for Large Scale Multiprocessors. In *International Symposium on High-Performance Computer Architecture*, pages 286–295, 1995.

[26] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Computer*, 28:690–691, 1979.

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

[28] 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 *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 77–90, 2010.

[29] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25:63–79, 1992.

[30] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *ACM Symposium on Principles of Programming Languages*, pages 378–391, 2005.

[31] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *ACM/IEEE International Symposium on Computer Architecture*, pages 289–300, 2008.

[32] M. Naik and A. Aiken. Conditional Must Not Aliasing for Static Race Detection. In *ACM Symposium on Principles of Programming Languages*, pages 327–338, 2007.

[33] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies Using Strata. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 229–240, 2006.

[34] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 97–108, 2009.

[35] M. Olszewski, J. Ansel, and S. Amarasinghe. Scaling Deterministic Multithreading. In *Workshop on Determinism and Correctness in Parallel Programming*, 2011.

[36] T. Onodera, K. Kawachiya, and A. Koseki. Lock Reservation for Java Reconsidered. In *European Conference on Object-Oriented Programming*, pages 559–583, 2004.

[37] M. S. Papamarcos and J. H. Patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *ACM/IEEE International Symposium on Computer Architecture*, pages 348–354, 1984.

[38] 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 *ACM Symposium on Operating Systems Principles*, pages 177–192, 2009.

[39] G. Pokam, C. Pereira, K. Danne, R. Kassa, and A.-R. Adl-Tabatabai. Architecting a Chunk-based Memory Race Recorder in Modern CMPs. In *IEEE/ACM International Symposium on Microarchitecture*, pages 576–585, 2009.

[40] M. C. Rinard and M. S. Lam. The Design, Implementation, and Evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20:483–545, 1998.

[41] M. Ronsse and K. De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 17:133–152, 1999.

[42] K. Russell and D. Detlefs. Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 263–272, 2006.

[43] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, 1996.

[44] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-Grain Access Control for Distributed Shared Memory. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, 1994.

[45] Standard Performance Evaluation Corporation. *SPECjbb2000 Documentation*, release 1.01 edition, 2001.

[46] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 15–26, 2011.

[47] C. von Praun and T. R. Gross. Object Race Detection. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 70–82, 2001.

[48] C. von Praun and T. R. Gross. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. In *ACM Conference on Programming Language Design and Implementation*, pages 115–128, 2003.

[49] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 205–214, 2007.

[50] D. Weeratunge, X. Zhang, and S. Jagannathan. Analyzing Multicore Dumps to Facilitate Concurrency Bug Reproduction. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 155–166, 2010.

[51] M. Xu, R. Bodik, and M. D. Hill. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *ACM/IEEE International Symposium on Computer Architecture*, pages 122–135, 2003.