OCTET: Capturing and Controlling Cross-Thread Dependences Efficiently *

Michael D. Bond† Milind Kulkarni‡ Man Cao† Minjia Zhang† Meisam Fathi Salmi†
Swarnendu Biswas† Aritra Sengupta† Jipeng Huang†

† Ohio State University
‡ Purdue University
mikebond@cse.ohio-state.edu, milind@purdue.edu,
{caoma,zhanminj,fathi,biswass,sengupta,huangjip}@cse.ohio-state.edu

Abstract

Parallel programming is essential for reaping the benefits of parallel hardware, but it is notoriously difficult to develop and debug reliable, scalable software systems. One key challenge is that modern languages and systems provide poor support for ensuring *concurrency correctness properties*—atomicity, sequential consistency, and multithreaded determinism—because all existing approaches are impractical. Dynamic, software-based approaches slow programs by up to an order of magnitude because capturing and controlling cross-thread dependences (i.e., conflicting accesses to shared memory) requires synchronization at virtually every access to potentially shared memory.

This paper introduces a new software-based concurrency control mechanism called OCTET that soundly captures cross-thread dependences and can be used to build dynamic analyses for concurrency correctness. OCTET achieves low overheads by tracking the *locality state* of each potentially shared object. Non-conflicting accesses conform to the locality state and require no synchronization; only conflicting accesses require a state change and heavyweight synchronization. This optimistic tradeoff leads to significant efficiency gains in capturing cross-thread dependences: a prototype implementation of OCTET in a high-performance Java virtual machine slows real-world concurrent programs by only 26% on average. A dependence recorder, suitable for record & replay, built on top of OCTET adds an additional 5% overhead on average. These results suggest that OCTET can provide a foundation for developing low-overhead analyses that check and enforce concurrency correctness.

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

Keywords concurrency correctness; dynamic analysis

1. Introduction

Software must become more concurrent to reap the benefits of the ever-increasing parallelism available in modern hardware. However, writing correct parallel programs on modern hardware is notoriously difficult. Widely used, high-performance programming languages provide shared memory and locks, which are hard to use to develop software that is both correct and scalable. Programmers must deal with data races, atomicity violations, deadlock, poor scalability, weak memory models, and nondeterminism.

To address the challenges of writing parallel programs, researchers have proposed a wide variety of language and system support to guarantee correct concurrent execution either by *enforcing* crucial correctness properties or by *checking* such properties and detecting when they are violated [5, 6, 10, 13–16, 18, 21, 23, 30–32, 34, 35, 37–39, 41, 45, 50, 52, 53]. For the rest of the paper, we refer to all such approaches as *analyses* for brevity. Examples include enforcing determinism via record & replay (e.g., DoublePlay [50]) and deterministic execution (e.g., CoreDet [6]); enforcing atomicity (transactional memory [21]); checking atomicity (e.g., Velodrome [18]); and checking sequential consistency and data-race freedom (e.g., DRFx [37]).

While these proposed analyses provide desirable correctness properties that ease the task of parallel programming, they are all *impractical* in various ways (Section 2.1). Many require expensive whole-program analysis or custom hardware support. A promising class of analyses are *dynamic*

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2374-1/13/10. . . $15.00.
http://dx.doi.org/10.1145/2509136.2509519

and *software-only*, avoiding both complex compiler analysis and specialized hardware. Unfortunately, such analyses suffer from very high overhead, slowing down applications by roughly an order of magnitude.

In this paper, we focus on the key cost that makes dynamic, software-only analyses expensive: the challenge of dealing with *cross-thread dependences*, the dependences that arise between threads as they access shared memory. Virtually all analyses rely, at their root, on soundly capturing cross-thread dependences and using this information to enforce or check concurrency correctness. However, because shared-memory programs can introduce cross-thread dependences on *any* shared-memory access, correctly detecting all such dependences requires existing software analyses to insert *synchronized* instrumentation at every access (unless the access can be proven to be well-synchronized). The synchronization overhead of this instrumentation is proportional to the number of program accesses to potentially shared data, and hence slows programs substantially.

This paper presents OCTET, a novel concurrency control mechanism that soundly captures all cross-thread dependences without heavyweight synchronized instrumentation. In fact, through an object-based, cache-coherence-like mechanism, OCTET's synchronization overheads are merely proportional to the number of cross-thread dependences in a program, rather than all accesses.

Contributions

The key insight undergirding OCTET is that most accesses to memory—even shared memory—exhibit thread locality. The same thread repeatedly accesses an object, or multiple threads perform only reads, so the accesses cannot introduce cross-thread dependences. By efficiently detecting (without synchronization) whether an access is definitely *not* involved in a cross-thread dependence, OCTET is able to avoid expensive synchronization in most cases, and only incurs high overhead when a cross-thread dependence occurs.

OCTET achieves these performance gains by implementing a cache-coherence-like system at the granularity of objects. Each potentially shared object has a "thread locality" state associated with it. This state expresses whether a thread can access the object without synchronization. A thread T can have write access ($\mathsf{WrEx_T}$), exclusive read access ($\mathsf{RdEx_T}$), or shared read access (RdSh); these states correspond to the M, E, and S states, respectively, in a MESI cache coherence protocol [44]. OCTET's instrumentation checks, before each object access, whether the access is compatible with the object's state. As in hardware cache coherence, if a thread tries to access an object that is already in a compatible state, no other work is necessary. There is no possibility of a cross-thread dependence, and OCTET needs no synchronization. Instead, synchronization is only necessary when a thread tries to access an object in an incompatible state. In such situations, OCTET uses various protocols to safely perform the state change before allowing the thread's access to proceed. Section 3 describes the design of OCTET in more detail, including elaborating on these state change protocols. Section 3.7 proves the soundness of the scheme (i.e., all cross-thread dependences are captured) and its liveness (i.e., OCTET does not deadlock).

We have implemented OCTET in a high-performance JVM; Section 4 describes the details. Section 5 evaluates the performance and behavior of our OCTET implementation on 13 large, multithreaded Java benchmark applications. We present statistics that confirm our hypotheses about the typical behavior of multithreaded programs, justifying the design principles of OCTET, and we demonstrate that our implementation of OCTET has overheads of 26% on average—significantly better than the overheads of prior mechanisms, and potentially low enough for production systems.

Because a variety of analyses rely on taking action upon the detection of cross-thread dependences, OCTET can serve as a foundation for designing new, efficient analyses. Section 6 discusses opportunities and challenges for implementing efficient analyses on top of OCTET. Section 7 presents an API for building analyses on top of OCTET, and we describe and evaluate how a low-overhead dependence recorder uses this API, suggesting OCTET can be a practical platform for new analyses.

2. Background and Motivation

Language and system support for concurrency correctness offers significant reliability, scalability, and productivity benefits. Researchers have proposed many approaches that leverage static analysis and language support (e.g., [9, 10, 39]) and custom hardware (e.g., [23, 35, 38]). Unfortunately, these techniques are impractical for contemporary systems: sufficiently precise static analyses do not scale to large applications; language-based approaches do not suffice for existing code bases; and hardware support does not exist in current architectures.

As a result of these drawbacks, there has been substantial interest in *dynamic*, *sound*,[1] *software-only* approaches guaranteeing concurrency correctness. Section 2.1 explains why software-based analyses suffer from impractically high overhead. Section 2.2 illuminates the key issue: the high overhead of capturing or controlling cross-thread dependences.

2.1 Dynamic Support for Concurrency Correctness

This section motivates and describes key analyses and systems, which we call simply *analyses*, that guarantee concurrency correctness properties by either enforcing or checking these properties at run time. For each analysis, we describe the drawbacks of prior work and then identify the key performance challenge(s).

Multithreaded record & replay. Record & replay of multithreaded programs provides debugging and systems ben-

[1] Following prior work, a dynamic analysis or system is *sound* if it guarantees no false negatives for the current execution.

efits. *Offline* replay allows reproducing production failures at a later time. *Online* replay allows running multiple instances of the same process simultaneously with the same interleavings, enabling systems benefits such as replication-based fault tolerance. Recording execution on multiprocessors is hard due to the frequency of potentially racy shared memory accesses, which require synchronized instrumentation to capture soundly [30]. *Chimera* rules out non-racy accesses using whole-program static analysis, which reports many false positives [31]. To achieve low overhead, Chimera relies on profiling runs to identify mostly non-overlapping accesses and expand synchronization regions.

Many multithreaded record & replay approaches sidestep the problem of capturing cross-thread dependences explicitly, but introduce limitations. Several support only online or offline replay but not both (e.g., [32, 45]). *DoublePlay* requires twice as many cores as the original program to provide low overhead, and it relies on data races mostly not causing nondeterminism, to avoid frequent rollbacks [50].

The key performance challenge for recording multithreaded execution is tracking cross-thread dependences.

Deterministic execution. An alternative to record & replay is executing multithreaded programs deterministically [6, 14, 34, 41]. As with record & replay, prior approaches avoid capturing cross-thread dependences explicitly in software because of the high cost. Existing approaches all have serious limitations: they either do not handle racy programs [41], add high overhead [6], require custom hardware [14], or provide per-thread address spaces and merge changes at synchronization points, which may not scale well to programs with fine-grained synchronization [34].

The performance challenge for multithreaded deterministic execution is controlling the ways in which threads interleave, i.e., controlling cross-thread dependences.

Guaranteeing sequential consistency and/or data race freedom. An execution is *sequentially consistent* (SC) if threads' operations appear to be totally ordered and respect program order [29]. An execution is *data-race-free* (DRF) if all conflicting accesses are ordered by the *happens-before* relationship [28], i.e., ordered by synchronization. Language memory models typically guarantee SC for DRF executions [1, 2, 36].

Sound and precise checking of SC or DRF is expensive (even with recent innovations in happens-before race detection [16]). An attractive alternative to checking SC or DRF is the following relaxed constraints: DRF executions must report no violation, and SC-violating executions must report a violation, but SC executions with races may or may not report a violation [19]. Recent work applies this insight by checking for conflicts between overlapping, synchronization-free regions [35, 37], but it relies on custom hardware to detect conflicting accesses efficiently.

The main performance challenge of detecting data races soundly is tracking cross-thread dependences. Precision is also important: the detector must maintain "when" variables were last accessed in order to avoid false positives.

Checking atomicity. An operation is *atomic* if it appears to happen all at once or not at all. Dynamic analysis can check that existing lock-based programs conform to an atomicity specification [18, 53]. *Velodrome* soundly and precisely checks *conflict serializability* (CS), a sufficient condition for atomicity, by constructing a region dependence graph that requires capturing cross-thread dependences [18]. It cannot check for atomicity violations in production because it slows programs by about an order of magnitude.

Similarly to detecting races, the performance challenges of detecting atomicity violations are tracking cross-thread dependences soundly and providing precision.

Enforcing atomicity. *Transactional memory* (TM) systems enforce programmer-specified atomicity annotations by speculatively executing atomic regions as *transactions*, which are rolled back if a region conflict occurs [21]. Custom-hardware-based approaches offer low overhead [20], but any real hardware support will likely be limited [5],[2] making efficient software TM (STM) an important long-term goal. Existing STMs suffer from two major, related problems—poor performance and weak semantics. Existing STM systems slow transactions significantly in order to detect conflicting accesses soundly. Furthermore, these systems typically provide only *weak atomicity* semantics because *strong atomicity* (detecting conflicts between transactions and non-transactional instructions) slows *all* program code substantially in order to detect conflicts between transactional and non-transactional code.

Achieving efficient, strongly atomic STM is challenging because of the cost of detecting and resolving cross-thread dependences throughout program execution. STMs also need some precision about when variables were last accessed (e.g., read/write sets [38]) in order to determine if a conflicting variable was accessed in an ongoing transaction.

In addition to these specific analyses, researchers have recently stressed the general importance of dynamic approaches for providing programmability and reliability guarantees and simplifying overly-complex semantics [1, 12].

2.2 Capturing and Controlling Cross-Thread Dependences

Despite its benefits, efficient software support for checking and enforcing concurrency correctness properties has remained elusive: existing software-based analyses slow programs significantly, often by up to an order of magnitude. What makes this diverse collection of analyses and systems so slow? As seen in the previous section, a recurring theme emerges: to capture concurrent behavior accurately, these

[2] http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/

```
// T1:                    // T2:
... = obj1.f;             obj2.f = ...;
```

Figure 1. Potential cross-thread dependence.

```
do {
  lastState = obj.state; // load per−object metadata
} while (lastState == LOCKED ||
         !CAS(&obj.state, lastState, LOCKED));
if (lastState != WrEx_T) {
  handlePotentialDependence(...);
}
obj.f = ...;   // program write
memfence;
obj.state = WrEx_T;  // unlock and update metadata
```

Figure 2. A write barrier that captures cross-thread dependences. Before the write, the per-object metadata obj.state should be WrEx_T (T is the current thread), or else there is a potential cross-thread dependence. LOCKED indicates the metadata is locked.

analyses must *capture cross-thread dependences*, two accesses executed by different threads that have any kind of data dependence: a true (write–read), anti (read–write), or output (write–write) dependence. Figure 1 shows a potential cross-thread dependence. Since this dependence arises only when obj1 and obj2 reference the same object, and when the threads interleave in a particular manner, it is hard to accurately predict this dependence statically.

Capturing cross-thread dependences typically involves adding barriers[3] that access and update shared metadata at each program read and write. Because program accesses may not be well synchronized (i.e., may have a data race), the barriers must use synchronization to avoid data races on the metadata accesses. Figure 2 shows how a barrier enforces atomic metadata accesses. It uses per-object metadata, annotated as obj.state, to track the last thread to access the metadata. The value of obj.state might be WrEx_T (write exclusive for thread T) where T is the last thread that wrote the object referenced by obj, or RdSh (read shared) if any thread last read the object. Tracking these per-object states is a natural way to capture cross-thread dependences; we use the notations obj.state, WrEx_T, and RdSh to closely match the notations used by our new approach (Section 3). If another thread accesses obj, it handles the the potential cross-thread dependence in an analysis-specific way and updates the state. The barrier locks the per-object metadata using an atomic compare-and-swap (CAS) instruction,[4] creating a small critical section while handling the potential dependence and performing the program memory access.

Our evaluation shows that adding such a barrier to all potentially shared memory accesses slows programs by 6.2X on average even without performing any analysis-specific operations, i.e., handlePotentialDependence() as a no-op (Section 5.4). These overheads are unsurprising because atomic instructions and memory fences serialize in-flight instructions, and atomic instructions invalidate cache lines.

The next section introduces OCTET, a system that captures cross-thread dependences efficiently. Section 6 describes how OCTET can be used to build new analyses that check and enforce concurrency correctness, and Section 7 sketches how OCTET can be used to build a low-overhead system for recording concurrent execution.

3. Capturing and Controlling Cross-Thread Dependences Efficiently

This section describes our approach for efficiently and accurately detecting or enforcing *cross-thread dependences* on shared objects:[5] data dependences involving accesses to the same variable by different threads. In prior work, capturing or controlling cross-thread dependences in software has proven expensive (Section 2).

We have designed a dynamic analysis framework called OCTET[6] to track cross-thread dependences at low overhead. OCTET's approach is based on a key insight: *the vast majority of accesses, even to shared objects, are not involved in a cross-thread dependence*. If OCTET can detect efficiently whether an access *cannot* create a cross-thread dependence, it can perform synchronization *only* when conflicting accesses occur, and dramatically lower the overhead of detecting cross-thread dependences.

To achieve this goal, OCTET associates a *thread-locality state* with each potentially shared object. This state describes which accesses will definitely not cause cross-thread dependences. These accesses proceed without synchronization or changing the object's state (the *fast path*), while accesses that imply a potential cross-thread dependence trigger a coordination protocol involving synchronization to change the object's state so that the access is permitted (the *slow path*). OCTET's synchronization costs are thus proportional to the number of *conflicting* accesses in a program, rather than all accesses or even shared accesses.

OCTET's state transitions provide two key features:

1. Each state transition results in the creation of a happens-before edge,[7] as shown in the following section. All cross-thread dependences are guaranteed to be ordered (transitively) by these happens-before edges (as proved in Section 3.7).

[3] A read (or write) barrier is instrumentation that executes before every read (write) [54].

[4] The atomic instruction CAS(addr, oldVal, newVal) attempts to update addr from oldVal to newVal, returning true on success. (We use the common term CAS, but our "CAS" acts like a *test-and-set* operation.)

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

[6] Optimistic cross-thread explicit tracking. OCTET "optimistically" assumes most accesses do not conflict and supports them at low overhead, at the cost of more expensive coordination when accesses conflict.

[7] Happens-before is a partial order on program and synchronization order [28]. Creating a happens-before edge means ensuring a happens-before ordering from the source to the sink of the edge.

Fast/slow path	Transition type	Old state	Access	New state	Synchronization needed	Cross-thread dependence?
Fast	Same state	$\mathsf{WrEx_T}$	R or W by T	Same	None	No
		$\mathsf{RdEx_T}$	R by T	Same		
		$\mathsf{RdSh_c}$	R by T if $\mathsf{T.rdShCount} \geq \mathsf{c}$	Same		
Slow	Upgrading	$\mathsf{RdEx_T}$	W by T	$\mathsf{WrEx_T}$	CAS	No
		$\mathsf{RdEx_{T1}}$	R by T2	$\mathsf{RdSh_{gRdShCount}}$		Potentially yes
	Fence	$\mathsf{RdSh_c}$	R by T if $\mathsf{T.rdShCount} < \mathsf{c}$	$(\mathsf{T.rdShCount} = \mathsf{c})$	Memory fence	Potentially yes
	Conflicting	$\mathsf{WrEx_{T1}}$	W by T2	$\mathsf{WrEx^{Int}_{T2}} \rightarrow \mathsf{WrEx_{T2}}$	Roundtrip coordination	Potentially yes
		$\mathsf{WrEx_{T1}}$	R by T2	$\mathsf{RdEx^{Int}_{T2}} \rightarrow \mathsf{RdEx_{T2}}$		
		$\mathsf{RdEx_{T1}}$	W by T2	$\mathsf{WrEx^{Int}_{T2}} \rightarrow \mathsf{WrEx_{T2}}$		
		$\mathsf{RdSh_c}$	W by T	$\mathsf{WrEx^{Int}_{T}} \rightarrow \mathsf{WrEx_{T}}$		

Table 1. OCTET state transitions fall into four categories that require different levels of synchronization.

2. Both fast-path state checks and slow-path state transitions execute atomically with their subsequent access. Client analyses can hook into the fast and/or slow paths without requiring extra synchronization, allowing them to perform actions such as updating analysis state or perturbing program execution.

The remainder of this section describes OCTET's operations.

3.1 OCTET States

A thread-locality state for an object tracked by OCTET captures access permissions for that object: it specifies which accesses can be made to that object without changing the state. We say that such accesses are *compatible with* the state. Accesses compatible with the state 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 changing the state. 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 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 thread-local counter: $\mathsf{T.rdShCount} \geq \mathsf{c}$ (Section 3.4).

Note the similarity of the thread-locality states to coherence states in the standard MESI cache-coherence protocol [44]. Modified corresponds to $\mathsf{WrEx_T}$, Exclusive corresponds to $\mathsf{RdEx_T}$, and Shared corresponds to $\mathsf{RdSh_c}$. Invalid corresponds to how threads *other than* T see $\mathsf{WrEx_T}$ or $\mathsf{RdEx_T}$ objects.

While the RdSh and RdEx states are not strictly necessary, their purpose is similar to E and S in the MESI protocol. Without the RdEx state, we have found that scalability degrades significantly, due to more RdSh→WrEx transitions.

We introduce additional *intermediate* states that help to implement the state transition protocol (Section 3.3):

$\mathsf{WrEx^{Int}_T}$: Intermediate state for transition to $\mathsf{WrEx_T}$.

$\mathsf{RdEx^{Int}_T}$: Intermediate state for transition to $\mathsf{RdEx_T}$.

3.2 High-Level Overview of State Transitions

OCTET inserts a write barrier before every write of a potentially shared object:

```
if (obj.state != WrEx_T) {
  /* Slow path: change obj.state & call slow-path hooks */
}
/* Call fast-path hooks */
obj.f = ...; // program write
```

and a read barrier before every read:

```
if (obj.state != WrEx_T && obj.state != RdEx_T &&
    !(obj.state == RdSh_c && T.rdShCount >= c)) {
  /* Slow path: change obj.state & call slow-path hooks */
}
/* Call fast-path hooks */
... = obj.f; // program read
```

When a thread attempts to access an object, OCTET checks the state of that object. If the state is compatible with the access, the thread takes the fast path and proceeds without synchronization; the overhead is merely the cost of the state check. Otherwise, the thread takes the slow path, which initiates a coordination protocol to change the object's state.

A key concern is what happens if another thread changes an object's state between a successful fast-path check and the access it guards. As Section 3.3 explains in detail, OCTET's protocols ensure that the other thread "coordinates" with the thread taking the fast path before changing the state, preserving atomicity of the check and access. A state change *without* coordination might miss a cross-thread dependence. (Imagine a cache coherence protocol that allowed a core to upgrade a cache line from Shared to Modified without waiting for other cores to invalidate the line in their caches!)

Table 1 overviews OCTET's behavior when a thread attempts to perform an access to an object in various states. As described above, when an object is already in a state that permits an access, no state change is necessary, while in other situations, an object's state must be changed.

State changes imply a potential cross-thread dependence, so OCTET must perform some type of coordination to ensure that the object's state is changed safely and a happens-before

edge is created to capture that potential dependence. OCTET uses different kinds of coordination to perform state changes, depending on the type of transition needed:

- Some transitions directly indicate the two or more threads involved in a potential dependence, as the transitions themselves imply a flow (write–read), anti (read–write), or output (write–write) dependence. These *conflicting* transitions require coordinating with other threads to establish the necessary happens-before relationships (Section 3.3).
- Other transitions do not directly indicate the threads involved in the dependence. For example, transitioning from $\mathsf{RdEx}_{\mathsf{T1}}$ to RdSh due to a read by T2 implies a potential dependence with the previous *writer* to an object. *Upgrading* transitions capture this dependence transitively by establishing a happens-before relationship between T1 and T2 (Section 3.4).
- Finally, when a thread reads an object in RdSh state, it must establish happens-before with the last thread to write that object, which might be any thread in the system. *Fence* transitions establish this happens-before transitively by ensuring that each read of a RdSh object happens after the object's transition to RdSh (Section 3.4).

Note that while a particular state change may not directly imply a cross-thread dependence, the combined set of happens-before edges created by OCTET ensures that every cross-thread dependence is ordered by these happens-before relationships. Section 3.7 proves OCTET's soundness, and Sections 6 and 7 describe how analyses can build on OCTET's happens-before edges, by defining hooks called from the slow path, in order to capture all cross-thread dependences soundly.

3.3 Handling Conflicting Transitions

When an OCTET barrier detects a conflict, 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 T2 changes the state of an object while another thread, T1, that has access to the object is between its state check and its access, then T1 and T2 may perform racy conflicting accesses without being detected—even if T2 uses synchronization.

At a high level, a thread—called the *requesting thread*—that wants to perform a conflicting state change *requests* access by sending a message to the thread—called the *responding thread*—that currently has access. The responding thread *responds* to the requesting thread when the responding thread is at a *safe point*, a point in the program that is definitely *not* between an OCTET barrier check and its corresponding access. (If the responding thread is blocked at a safe point, the response happens *implicitly* when the requesting thread observes this fact atomically.) Upon receiving the response, the requesting thread can change the object's state and proceed with its access. This roundtrip coordination between threads results in a happens-before relationship being established between the responding thread's *last* access to the object and the requesting thread's access to the object, capturing the possible cross-thread dependence.

To handle RdSh→WrEx transitions, which involve multiple responding threads, the requesting thread performs the coordination protocol with each responding thread.

Safe points. OCTET distinguishes between two types of safe points: *non-blocking* and *blocking*. Non-blocking safe points occur during normal program execution (e.g., at loop back edges); at these safe points, the responding thread checks for and responds to any requests *explicitly*. Blocking safe points occur when the responding thread is blocked (e.g., while waiting to acquire a lock, or during file I/O). In this scenario, the responding thread cannot execute code, so it instead *implicitly* responds to any requests by setting a per-thread flag that requesting threads can synchronize with.

Safe point placement affects both correctness and performance. To provide atomicity of OCTET instrumentation and program accesses, safe points *cannot* be between a state check (or state transition) and its corresponding access. To avoid deadlock, any point where a thread might block needs a safe point, and code must not be able to execute an unbounded number of steps without executing a safe point. Increasing this bound but keeping it finite would hurt performance but not correctness. It suffices to place non-blocking safe points at loop back edges and method entries, and treat all potentially blocking operations as blocking safe points. Language VMs typically already place non-blocking safe points at the same points (e.g., for timely GC), and threads enter a special blocking state at potentially blocking operations (e.g., to allow GC while blocked).

Request queues. Conceptually, every thread maintains a *request queue*, which serves as a shared structure for coordinating interactions between threads. The request queue for a (responding) thread respT allows other (requesting) threads to signal that they desire access to objects that respT currently has access to (i.e., objects in $\mathsf{WrEx}_{\mathsf{respT}}$, $\mathsf{RdEx}_{\mathsf{respT}}$, or RdSh states). The queue is also used by respT to indicate to any requesting threads that respT is at a blocking safe point, implicitly relinquishing ownership of any requested objects.

The request queue interface provides several methods. The first four are called by the responding thread respT:

requestsSeen() Returns true if there are any pending requests for objects owned by respT.

handleRequests() Handles and responds to pending requests. Performs memory fence behavior.

handleRequestsAndBlock() Handles requests as above, and atomically[8] places the request queue into a "blocked" state indicating that respT is at a blocking safe point.

resumeRequests() Atomically unblocks the queue.

[8] An atomic operation includes memory fence behavior.

At non-blocking safe points:

```
1  if (requestsSeen())
2    handleRequests();
```

At blocking safe points:

```
3  handleRequestsAndBlock();
4  /* blocking actions */
5  resumeRequests();
```

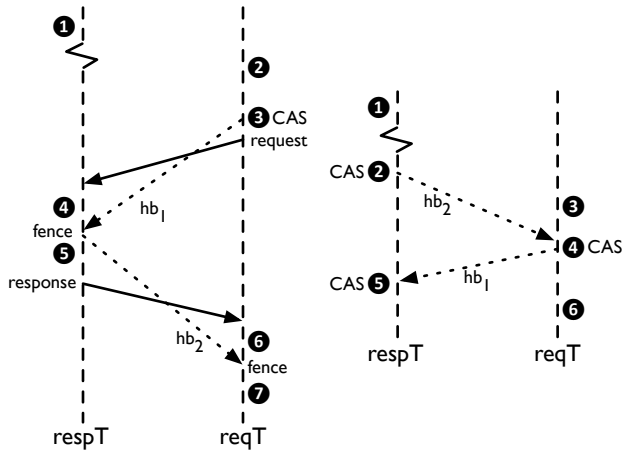
(a) Responding thread (thread respT)

To move obj from $\text{WrEx}_{\text{respT}}$ or $\text{RdEx}_{\text{respT}}$ to $\text{WrEx}_{\text{reqT}}$:

```
6  currState = obj.state; // WrEx_respT or RdEx_respT expected
7  while (currState == any intermediate state ||
8         !CAS(&obj.state, currState, WrEx^Int_reqT)) {
9    // obj.state ← WrEx^Int_reqT failed
10   // non-blocking safe point:
11   if (requestsSeen()) { handleRequests(); }
12   currState = obj.state;  // re-read state
13 }
14 handleRequestsAndBlock(); // start blocking safe point
15 response = request(getOwner(currState));
16 while (!response) {
17   response = status(getOwner(currState));
18 }
19 resumeRequests(); // end blocking safe point
20 obj.state = WrEx_reqT;
21 /* proceed with access */
```

(b) Requesting thread (thread reqT)

Figure 3. Protocol for conflicting state changes.



(a) Explicit protocol: (1) respT accesses obj; (2) reqT puts obj in $\text{WrEx}^{\text{Int}}_{\text{reqT}}$ (line 8); (3) reqT issues request to respT with CAS (line 15); (4) respT receives request (line 1). (5) respT issues a fence, establishing hb_1, and responds (line 2); (6) reqT sees response and unblocks, which includes fence behavior (lines 16–19), establishing hb_2; (7) reqT places obj in $\text{WrEx}_{\text{reqT}}$ and performs write (lines 20–21).

(b) Implicit protocol: (1) respT accesses obj; (2) respT blocks with CAS (line 3); (3) reqT places obj in $\text{WrEx}^{\text{Int}}_{\text{reqT}}$ (line 8) and blocks (line 14); (4) reqT issues a request to respT with a CAS and observes that it is blocked, establishing hb_2 (line 15); (5) respT unblocks with CAS, establishing hb_1 (line 5); (6) reqT places obj in $\text{WrEx}_{\text{reqT}}$ and performs write (lines 20–21).

Figure 4. Operation of (a) explicit and (b) implicit coordination protocols for conflicting transitions. Line numbers refer to Figure 3.

The next two methods are called by the requesting thread reqT and act on the queue of the responding thread respT:

request(respT) Makes a request to respT with a CAS. Returns true if reqT's queue is in the blocked state.

status(respT) Returns true if reqT's request has been seen and handled by respT.

Appendix C describes a concrete implementation of these methods.

Figure 3 shows the protocol for handling conflicting accesses.[9] Figure 4 shows how both the explicit and implicit protocols establish happens-before relationships between threads. To see how the request queue is used to coordinate access to shared objects, consider the case where respT has access to object obj in $\text{WrEx}_{\text{respT}}$ or $\text{RdEx}_{\text{respT}}$ state, and reqT wants to write to the object (reads work similarly).

Requesting thread. reqT first atomically places obj into the desired final state with an *intermediate* flag set (line 8 in Figure 3(b)). This intermediate flag indicates to all threads that this object is in the midst of a state transition, preventing other attempts to access the object or change its state until the coordination process completes. We refer to any state with the intermediate flag set as an "intermediate state." The use of intermediate states prevents races in the state transition protocol (obviating the need for numerous transient states, as exist in cache coherence protocols). Note that reqT's CAS may fail, if a third thread simultaneously attempts to access obj. Before reqT retries its CAS (lines 7–13 in Figure 3(b)), it responds to any pending requests on *its* request queue, to avoid deadlock (line 11 in Figure 3(b)). After placing obj into an intermediate state, reqT adds a request to respT's request queue by invoking request(respT) (line 15 in Figure 3(b)).

Responding thread. To *respond* to a request, respT uses either the *explicit* or *implicit* response protocol. When respT is at a non-blocking safe point, it can safely relinquish control of objects using the explicit protocol. respT checks if there are any pending requests on its request queue and calls handleRequests to deal with them (lines 1–2 in Figure 3(a)). Because handleRequests performs a fence, the explicit response protocol establishes two happens-before relationships (Figure 4(a)): (1) between reqT's initial request for access to obj and any subsequent attempt by respT to access obj (respT will then find obj in $\text{WrEx}^{\text{Int}}_{\text{reqT}}$ state) and (2) between respT's response to reqT and reqT's access to obj.

The implicit response protocol is used when respT is at a *blocking* safe point. Intuitively, if reqT knows that respT is at a blocking safe point, it can assume a response implicitly. Before blocking, respT sets its request queue as "blocking," letting reqT see that respT is at a blocking safe point (line 3

[9] Note that our pseudocode mixes Java code with memory fences (memfence) and atomic operations (CAS). The JVM compiler must treat these operations like other synchronization operations (e.g., lock acquire and release) by not moving loads and stores past these operations.

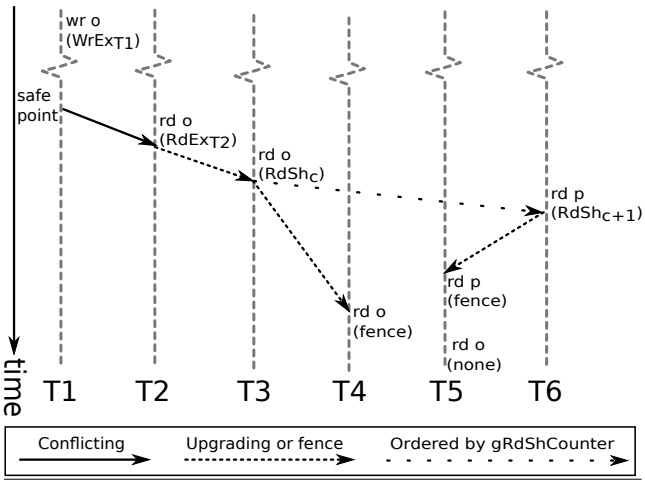


Figure 5. Example execution illustrating upgrading and fence transitions. After T1 writes o, o is in the WrEx_{T1} state. Then T2 reads o, triggering a conflicting transition to RdEx_{T2}. Next, T3 reads o, triggering an upgrading transition to RdSh_c. T4 then reads o, triggering a fence transition, assuming it has *not* already read an object with state RdSh_{c+1}. T5 reads o, but the read does *not* trigger a fence transition because T5 already read an object p that T6 transitioned to RdSh_{c+1}.

in Figure 3(a)). This protocol establishes a happens-before relationship from reqT to respT as in the explicit protocol. Further, it establishes a relationship between respT's *entering the blocked state* and reqT's access to obj (Figure 4(b)).

3.4 Handling Upgrading and Fence Transitions

Accesses may be non-conflicting but still require a state change. These *upgrading* and *fence* transitions do not need the coordination protocol, but they do require synchronization in order to avoid state-change races and establish happens-before ordering.

***Upgrading from* RdEx.** A write by T1 to a RdEx_{T1} object triggers an *upgrading* transition to WrEx_{T1}, which requires an atomic instruction in order to avoid races with other threads changing the object's state. This transition does not need to establish any happens-before edges, since any cross-thread dependences will be implied by the happens-before edges added by other transitions.

A read by T3 to a RdEx_{T2} object, as in Figure 5, triggers an upgrading transition to RdSh state. Note that the coordination protocol is not needed here because it is okay for T2 to continue reading the object after its state changes.

In order to support fence transitions (described next), OCTET globally orders all transitions to RdSh states. To do so, it increments a global counter gRdShCount atomically at each transition to RdSh and uses the incremented value as c in the new state RdSh_c.

Fence transitions. When a thread reads an object o in RdSh_c state, there is a potential dependence with the last thread to *write* to o. The value of c in o's state establishes that any write must have happened *before* gRdShCount was incremented to c. Each thread T has a counter T.rdShCount that indicates the last time the thread synchronized with gRdShCount. If T.rdShCount $\geq$ c, then the thread synchronized recently enough that the read must be after the write. Otherwise, the slow path issues a fence and updates T.rdShCount, establishing the necessary happens-before.

To see how this protocol works, consider the actions of threads T4–T6 in Figure 5. T4 reads o in the RdSh_c state. To capture the write–read dependence, T4 checks if T4.rdShCount $\geq$ c. If not, T4 triggers a *fence* transition to ensure that it reads o *after* o was placed in RdSh_c state. This transition issues a load fence to ensure a happens-before relationship with o's transition to RdSh_c by T3, and updates T4.rdShCount $\leftarrow$ c.

T5's read of o does *not* trigger a fence transition because T5 already read p in the RdSh_{c+1} state and c+1 $>$ c. The write–read dependence is captured transitively by the happens-before edge on gRdShCount from T3 to T6 and the fence-transition-based happens-before edge from T6 to T5.

3.5 Correctness Assumptions

OCTET introduces additional synchronization into programs when cross-thread dependences arise. Hence, it is critical that this new synchronization does not introduce the possibility of deadlock or livelock into a program that would otherwise be free of such pathologies. In other words, there must never be a scenario where all threads are stuck at OCTET barriers and are unable to continue; at least one thread must be able to complete its access. OCTET's protocols make the following two assumptions:

1. The thread scheduler is *fair*; no thread can be descheduled indefinitely. This assumption holds true on most systems.
2. Attempts to add requests to a thread's request queue will eventually succeed. This assumption is true in practice for most concurrent queue implementations, given the first assumption.

Note that no other assumptions are needed (e.g., we need not assume that CASing objects into the intermediate state succeed). Under these conditions, we can show:

Theorem 1. OCTET*'s protocol is deadlock and livelock free.*

Proof. The proof of Theorem 1 is in Appendix A.1.

Another assumption is that shared counters such as gRdShCount and request counters (Appendix C) will not overflow. An implementation could reset counters periodically; tolerate overflow with a wraparound-aware design; or use 64-bit counters, making overflow unlikely.

3.6 Scalability Limitations

Our current design and implementation of OCTET have several limitations related to scalability. We discuss them here and suggest potential opportunities for future work.

OCTET magnifies the cost of program conflicts since they require a roundtrip coordination protocol. OCTET can add high overhead to programs with many conflicting transitions. Future work could extend the OCTET protocol with new "pessimistic" states for highly conflicted objects, which would require synchronization on every access but never require roundtrip coordination.

Some programs partition large arrays into disjoint chunks that threads access independently. The current OCTET design and implementation assign each array a single state, which would lead to many false conflicts in such programs. A future implementation could divide arrays into chunks of k elements each and assign each chunk its own OCTET state.

The global RdSh counter (gRdShCount) may be a scalability bottleneck since only one RdEx→RdSh transition can update it at once. A potential solution is to deterministically map each object to a counter in a modestly sized *array* of RdSh counters (many objects map to each counter), allowing multiple RdEx→RdSh transitions to proceed in parallel.

RdSh→WrEx transitions require roundtrip coordination with all threads. Future work can explore strategies including the following. (1) The RdSh counter could help identify threads that have definitely *not* read a given RdSh_c object (if $\mathsf{T.rdShCount} < \mathsf{c}$). This optimization would be more precise with the multiple RdSh counters described above. (2) A new "multi-read exclusive" state could track multiple, known reader threads, limiting RdSh→WrEx transitions.

3.7 Soundness of OCTET

OCTET is *sound*: it correctly creates happens-before relationships between all cross-thread dependences, and allows actions to be taken whenever such cross-thread dependences are detected. This behavior is the foundation of the various analyses that might be built on top of OCTET (Section 6).

Theorem 2. OCTET *creates a happens-before relationship to establish the order of every cross-thread dependence.*

Proof. The proof of Theorem 2 is in Appendix A.2.

At a high level, the proof proceeds by showing that for every possible cross-thread dependence, the various OCTET coordination and synchronization protocols ensure that there is a happens-before edge between the source and the sink of that dependence. Crucially, many cross-thread dependences are not directly implied by a single OCTET state transition but must instead be established transitively. For example, if T1 writes an object, then T2, T3, and T4 read that object in succession, there is a cross-thread read-after-write dependence between T1 and T4 even though T4 finds the object in RdSh state. The happens-before edge is established by a combination of conflicting, upgrading, and fence transitions.

By synchronizing on all cross-thread dependences, OCTET provides sequential consistency with respect to the *compiled* program, even on weak hardware memory models.

4. Implementation

We have implemented a prototype of OCTET in Jikes RVM 3.1.3, a high-performance Java virtual machine [3] that performs competitively with commercial JVMs.[10] OCTET is publicly available on the Jikes RVM Research Archive.[11]

OCTET*'s instrumentation.* Jikes RVM uses two dynamic compilers to transform bytecode into native code. The *baseline* compiler compiles each method the first time the method executes, translating bytecode directly to native code. The *optimizing* compiler recompiles hot (frequently executed) methods at increasing levels of optimization. We modify both compilers to add barriers at every read and write to an object field, array element, or static field. The compilers do not add barriers at accesses to final (immutable) fields, nor to a few known immutable classes such as String and Integer. The modified compilers add barriers to application methods and Java library methods (e.g., java.*).

A significant fraction of OCTET's overhead comes from barriers in the libraries, which must be instrumented in order to capture cross-thread dependences that occur within them. Since Jikes RVM is written in Java, both the application and VM call the libraries. We have modified Jikes RVM to compile two versions of each library method: one called from the application context and one called from the VM context. The compilers add OCTET barriers only to the version called from the application context.

OCTET*'s metadata.* To track OCTET states, the implementation adds one metadata word per (scalar or array) object by adding a word to the header. For static fields, it inserts an extra word per field into the global table of statics. Words are 32 bits because our implementation targets the IA-32 platform. The implementation represents WrEx_T, RdEx_T, $\mathsf{WrEx}_\mathsf{T}^\mathsf{Int}$, and $\mathsf{RdEx}_\mathsf{T}^\mathsf{Int}$ as the address of a thread object T and uses the lowest two bits (available since objects are word aligned) to distinguish the four states. The most common state, WrEx_T, is represented as T. The implementation represents RdSh_c as simply c, using a range that does not overlap with thread addresses: [0xc0000000, 0xffffffff].

Jikes RVM already reserves a register that always points to the current thread object (T), so checking whether a state is WrEx_T or RdEx_T is fast. To check whether an object's state is RdSh_c *and* T.rdShCount is up-to-date with c, the barrier compares the state with c. The implementation actually *decrements* the gRdShCount and T.rdShCount values so that a single comparison can check that T.rdShCount is up-to-date with respect to c (without needing to check that

[10] http://dacapo.anu.edu.au/regression/perf/

[11] http://www.jikesrvm.org/Research+Archive

the state is also RdSh). The implementation thus adds the following check before writes:

```
if (o.state != T) { /* slow path */ }
```

And it adds the following check before reads:

```
if ((o.state & ~0x1) != T &&
    o.state <unsigned T.rdShCount) { /* slow path */ }
```

The implementation initializes the OCTET state of newly allocated objects and newly initialized static fields to the $\mathsf{WrEx}_{\mathsf{T}}$ state, where T is the allocating/resolving thread. (The implementation could, but does not currently, handle another thread seeing an uninitialized state—guaranteed to be zero [36]—by waiting for the state to get initialized.)

Static optimizations. To some extent, OCTET obviates the need for static analyses that identify accesses that cannot conflict, because it adds little overhead at non-conflicting accesses. However, adding even lightweight barriers at every access adds nontrivial overhead, which we can reduce by identifying accesses that do not require barriers.

Some barriers are "redundant" because a prior barrier for the same object guarantees that the object will have an OCTET state that does not need to change. We have implemented an intraprocedural static analysis that identifies and removes redundant barriers at compile time. Appendix B describes and evaluates this analysis in more detail.

Alternative implementations. Although we implement OCTET inside of a JVM, alternative implementations are possible. Implementing OCTET in a dynamic bytecode instrumentation tool (e.g., RoadRunner [17]) would make it portable to any JVM, but it could be hard to make certain low-level features efficient, such as per-object metadata and atomic operations. One could implement OCTET for native languages such as C/C++ by modifying a compiler. A native implementation would need to (1) add safe points to the code and (2) handle per-variable metadata differently, e.g., by mapping each chunk of k bytes to an OCTET metadata word via shadow memory [40].

5. Evaluation

This section argues that OCTET is a suitable platform for capturing cross-thread dependences by showing that its optimistic tradeoff is worthwhile for real programs. We evaluate this tradeoff by measuring OCTET's run-time characteristics and performance, and comparing with two alternatives: a purely pessimistic approach and a prior optimistic approach.

5.1 Methodology

Benchmarks. The experiments execute our modified Jikes RVM on the parallel DaCapo Benchmarks [8] versions 2006-10-MR2 and 9.12-bach (2009) including a fixed lusearch [55], and fixed-workload versions of SPECjbb2000 and SPECjbb2005 called pjbb2000 and pjbb2005.[12] Table 2 shows the number of application threads that each program executes: total threads executed (Column 1) and maximum threads running at any time (Column 2). DaCapo benchmark names from the 2006 and 2009 versions have suffixes of 6 and 9, respectively. Some benchmarks by default spawn threads based on the number of available cores.

	Total threads	Max live threads
eclipse6	17–18	11–12
hsqldb6	402	60–102
lusearch6	65	65
xalan6	9	9
avrora9	27	27
jython9	3	2–3
luindex9	2	2
lusearch9	# cores	# cores
pmd9	5	5
sunflow9	# cores×2	# cores
xalan9	# cores	# cores
pjbb2000	37	9
pjbb2005	9	9

Table 2. The total number of threads executed by each program and the maximum number that are running at any time. Some values are ranges due to run-to-run nondeterminism.

A potential threat to validity is if the benchmarks, which are derived from real programs, are not representative of real-world parallel programs. Recent work suggests that some DaCapo benchmarks perform limited communication [24], which may or may not make them representative of real-world program behavior.

Platform. Experiments execute on an AMD Opteron 6272 system with 4 16-core processors running Linux 2.6.32. Unless stated otherwise, experiments run on 32 cores (using the taskset command) due to an anomalous result with 64 cores.

Measuring performance. To account for run-to-run variability due to dynamic optimization guided by timer-based sampling, we execute 25 trials for each performance result and take the median (to reduce the impact of performance outliers due to system noise). We also show the mean, as the center of 95% confidence intervals.

We build a high-performance configuration of Jikes RVM (FastAdaptiveGenImmix) that optimizes the VM ahead of time and adaptively optimizes the application at run time; it uses the default high-performance garbage collector and chooses heap sizes adaptively. Execution times naturally include dynamic compilation costs.

5.2 State Transitions

Table 3 shows the number of OCTET transitions executed, including accesses that do not change the state (i.e., fast path only). The three groups of columns show increasing levels of synchronization that correspond to the transitions in Table 1. *Alloc* counts the number of objects allocated (and static fields initialized); for each such event, the allocating thread T initializes an OCTET metadata word to $\mathsf{WrEx}_{\mathsf{T}}$.

[12] http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005

	Alloc or same state				Upgrading or fence			Conflicting			
	Alloc	WrEx	RdEx	RdSh	RdEx→WrEx	RdEx→RdSh	RdSh→RdSh	WrEx→WrEx	WrEx→RdEx	RdEx→WrEx	RdSh→WrEx
eclipse6	1.2×10^{10} (**99.9983**%)				7.0×10^{4} (**.00056**%)			1.4×10^{5} (**.0012**%)			
	2.6%	86%	2.0%	9.4%	.000078%	.00048%	.0000038%	.000043%	.00089%	.000035%	.00019%
hsqldb6	6.5×10^{8} (**99.78**%)				5.5×10^{5} (**.083**%)			8.9×10^{5} (**.14**%)			
	6.5%	88%	.45%	4.7%	.056%	.020%	.0082%	.045%	.078%	.00069%	.013%
lusearch6	2.5×10^{9} (**99.99971**%)				2.8×10^{3} (**.00011**%)			4.5×10^{3} (**.00018**%)			
	3.1%	91.0%	4.9%	1.0%	.000071%	.0000040%	.000037%	.0000085%	.00017%	0%	.0000027%
xalan6	1.1×10^{10} (**99.77**%)				9.9×10^{6} (**.091**%)			1.5×10^{7} (**.14**%)			
	2.2%	86%	.13%	11%	.091%	.000016%	.000030%	.049%	.091%	.0000000037%	.0000013%
avrora9	6.1×10^{9} (**99.80**%)				6.2×10^{6} (**.10**%)			6.1×10^{6} (**.099**%)			
	1.2%	89%	8.3%	1.4%	.018%	.014%	.069%	.046%	.037%	.0037%	.013%
jython9	5.4×10^{9} (**99.9999970**%)				5.8×10^{1} (**.0000011**%)			1.0×10^{2} (**.0000019**%)			
	6.2%	91.3%	.0022%	2.4%	.00000020%	.00000084%	.000000037%	0%	.0000019%	0%	0%
luindex9	3.5×10^{8} (**99.99983**%)				2.1×10^{2} (**.000062**%)			3.8×10^{2} (**.00011**%)			
	3.9%	96.0%	.078%	.012%	.000052%	.0000093%	.00000029%	.00000058%	.00011%	.00000029%	.0000012%
lusearch9	2.4×10^{9} (**99.99977**%)				2.6×10^{3} (**.00011**%)			2.9×10^{3} (**.00012**%)			
	3.1%	95.8%	.00046%	1.1%	.000025%	.000023%	.000058%	.0000033%	.00011%	.0000014%	.0000082%
pmd9	6.3×10^{8} (**99.988**%)				3.6×10^{4} (**.0057**%)			4.2×10^{4} (**.0067**%)			
	11%	86%	.062%	3.2%	.0040%	.00083%	.00093%	.00025%	.0063%	.0000018%	.00017%
sunflow9	1.7×10^{10} (**99.99986**%)				1.8×10^{4} (**.00010**%)			6.9×10^{3} (**.000040**%)			
	1.3%	40%	.0060%	59%	.0000010%	.000029%	.000073%	.0000067%	.000031%	0%	.0000017%
xalan9	1.0×10^{10} (**99.70**%)				1.2×10^{7} (**.12**%)			1.9×10^{7} (**.18**%)			
	2.8%	90%	.18%	7.1%	.12%	.000062%	.00026%	.065%	.12%	.000000011%	.000017%
pjbb2000	1.8×10^{9} (**99.90**%)				9.1×10^{5} (**.050**%)			9.5×10^{5} (**.052**%)			
	5.1%	78%	1.6%	16%	.044%	.0055%	.000085%	.00030%	.051%	.0000027%	.000082%
pjbb2005	6.9×10^{9} (**98.8**%)				3.5×10^{7} (**.51**%)			5.0×10^{7} (**.72**%)			
	4.2%	89%	1.9%	3.3%	.21%	.087%	.21%	.21%	.37%	.061%	.085%

Table 3. OCTET state transitions, including fast-path-only barriers that do not change the state. Conflicting transitions involve a temporary transition to an intermediate state ($\mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Int}}$ or $\mathsf{RdEx}_{\mathsf{T}}^{\mathsf{Int}}$). For each program, the first row is the sum of the column group, and the second row breaks down each transition type as a percentage of all transitions. We round each count to two significant digits, and each percentage x as much as possible such that x and $100\%-x$ each have at least two significant digits.

The table shows that the vast majority of accesses do not require synchronization. Lightweight fast-path instrumentation handles these transitions. *Upgrading and fence* transitions occur roughly as often as *Conflicting* transitions; the conflicting transitions are of greater concern because they are more expensive. Conflicting transitions range from fewer than 0.001% of all transitions for a few benchmarks, to 0.10–0.72% for a few benchmarks (xalan6, avrora9, xalan9, pjbb2005). The relative infrequency of conflicting transitions provides justification for OCTET's optimistic design.

5.3 Performance

Figure 6 presents the run-time overhead OCTET adds to program execution. Each bar represents overall execution time, normalized to unmodified Jikes RVM (*Base*).

The configuration *Octet w/o coordination* adds OCTET states and barriers, but does not perform the coordination protocol, essentially measuring only OCTET's fast-path overhead. (This configuration still performs state transitions; otherwise fast-path checks repeatedly fail, slowing execution.) OCTET's fast path adds 13% overhead on average.

Octet performs the coorindation protocol and adds 13% overhead on average over *Octet w/o coordination*. Overall OCTET overhead is 26% on average. Unsurprisingly, the coordination protocol adds more overhead to programs with higher fractions of conflicting transitions (hsqldb6, xalan6, avrora9, xalan9, and pjbb2005). The program pjbb2005 adds the most overhead and has the highest fraction of conflicting transitions. Interestingly, pjbb2005 has especially low overhead for approaches that are less optimistic (more pessimistic), even though these approaches perform poorly overall (Sections 5.4 and 5.5), showing the potential for an adaptive approach that converts roundtrip coordination operations to atomic or fence operations (Section 3.6). Despite hsqldb6's high fraction of conflicting transitions, its overhead is lower than other high-conflict programs because more of its conflicting transitions use the implicit protocol.

Scalability. As mentioned earlier, these experiments run on only 32 cores. We find that OCTET's overhead is lower running on 64 cores (20% on average), primarily because of cases like xalan9, on which *Octet* consistently *outperforms Base*. Investigating this anomalous result, we have found that simply modifying Jikes RVM (without OCTET) to insert a barrier before every access that does useless work (5–40 empty loop iterations) causes xalan9 to run faster on 64 cores (with the best performance for 20 loop iterations). This result suggests that the OCTET configurations improve performance simply by slowing down reads and writes, rather than anything specifically related to OCTET. We suspect that the Opteron 6272 machine we use is experiencing oversaturation on the interconnect that links together its 8 NUMA nodes (with 8 cores each). Although we have not been able

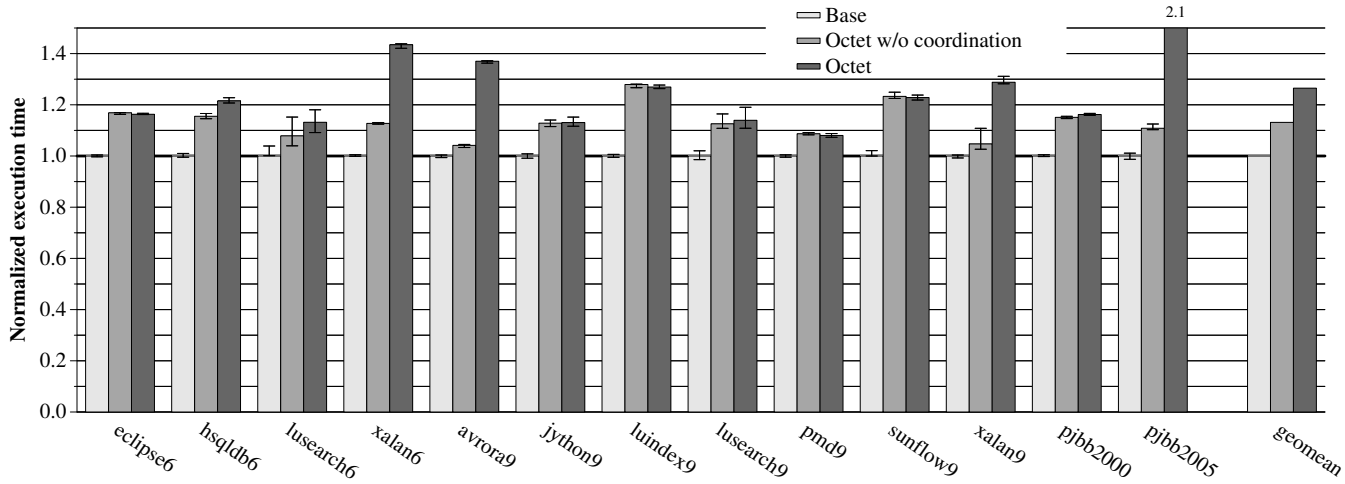


Figure 6. OCTET run-time performance. The ranges are 95% confidence intervals centered at the mean.

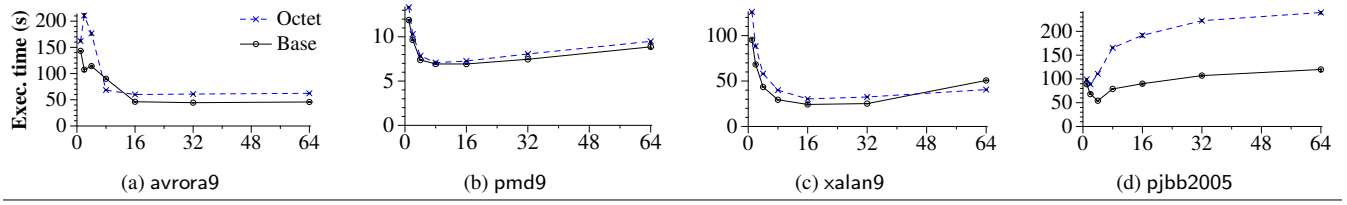


Figure 7. Scalability of Jikes RVM with and without OCTET on 1–64 cores (the x-axis) for four representative programs.

to fully understand this issue, our investigations demonstrate that the effect is not specific to OCTET but rather an effect of Jikes RVM running xalan9 on this architecture.

Figure 7 shows how the execution times of unmodified Jikes RVM (*Base*) and OCTET vary with different numbers of cores. Each point is the median of 10 trials, with 95% confidence intervals (mostly too short to see) centered around the mean. Although OCTET adds fairly low overhead to avrora9 for 8–64 cores, it slows the program by up to 2X on 2–4 cores for the following reason. When avrora9—which always runs 27 threads (Table 2)—runs on few cores, the OS thread scheduler maps several threads to the same core. A descheduled thread may or may not be blocked at a safe point; if not, the thread cannot respond to coordination protocol requests until it is scheduled. This issue could be mitigated by implementing better cooperation between the JVM and the OS thread scheduler, e.g., descheduling threads only at safe points. (In contrast, hsqldb6 has more threads than cores, but its threads block frequently, so descheduled threads are often blocked at a safe point.)

Figure 7(b) shows that OCTET adds fairly consistent overhead to pmd9 for 1–64 cores. With or without OCTET, Jikes RVM running pmd9 on this architecture stops scaling around 8 cores (Table 2 shows pmd9 executes only 5 threads). For 16–64 cores, run time increases; we find most of this effect is due to increasing GC time. We suspect that the parallel GC does not scale well on the NUMA architecture. The nine programs *not* shown in these plots show scalability similar to pmd9: OCTET adds fairly consistent overhead regardless of the number of cores, while each program scales differently depending on its thread count and inherent parallelism. Several of these programs scale better than pmd9: lusearch6, xalan6, lusearch9, and sunflow9.

Figure 7(c) shows that xalan9 scales up to 16–32 cores, and OCTET adds fairly consistent overhead. For 64 cores, we observe the anomaly described above in which OCTET actually outperforms unmodified Jikes RVM. Figure 7(d) shows that OCTET slows pjbb2005 by about 2X for 4–64 cores (due to conflicting transitions, as described previously). For 8–64 cores, execution time increases with the number of cores, which we believe is caused by pjbb2005's 9 threads (Table 2) being dispersed across more cores, increasing the effects of remote cache misses and NUMA communication.

5.4 Comparison to Pessimistic State Model

We have implemented and evaluated the "pessimistic" state model used for Figure 2 using the same implementation framework and experimental setup as for OCTET. Every memory access requires instrumentation that performs an atomic operation such as CAS. For simplicity of implementation, our barriers execute the program's write *outside* of the atomic region, an optimization that is acceptable for analyses that capture only *unordered* conflicting accesses. We find that adding the barrier to all potentially shared memory

accesses slows programs by 4.5–6.2X: the geomean is 4.5X excluding sunflow9 or 6.2X including sunflow9, which the pessimistic barriers slow by 345X (±4X). OCTET not only outperforms the pessimistic model on average, but is faster on all benchmarks except pjbb2005, which pessimistic barriers slow down by only 1.3X. Pessimistic barriers slow every other program by at least 2X.

We find that accesses to sunflow9's RdSh objects have little same-thread locality, i.e., threads interleave reads to each RdSh object, so performing a write to each read-shared access leads to numerous remote cache misses. In contrast, pessimistic barriers perform well for pjbb2005 because nearly all of its RdSh accesses have same-thread locality, so performing a write at each read-shared access does not add many remote cache misses.

5.5 Comparison to Alternative State Transition Model

For comparison purposes, we have implemented a state transition model from prior work. Von Praun and Gross describe an approach for detecting races based on tracking thread ownership of objects [51]. Their ownership system dynamically identifies shared objects, allowing the race detector to restrict its attention to those objects. Like OCTET, their approach uses unsynchronized checks, with requests and polling for state changes. 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, and objects that enter shared states cannot return to an exclusive state. In contrast, OCTET supports transitioning back to exclusive states, and accesses require synchronization only on state changes.

We have implemented von Praun and Gross's state transition model with as many similarities to the OCTET implementation as possible. We adapt their model, which is designed for detecting races with the lockset algorithm, to address the problem of tracking dependences. Our adapted model, which avoids lockset operations and some state transitions, should be no more expensive than the original model. An object starts in an exclusive state for its allocating thread; an access by a second thread triggers a conflicting transition to an exclusive state for the second thread; any subsequent cross-thread access triggers a conflicting transition to a shared–read or shared–modified state in which all accesses require an atomic operation such as a CAS. Shared–read and shared–modified objects cannot transition back to exclusive states. The model thus avoids conflicting transitions but can require a CAS for repeated accesses to shared objects.

Table 4 summarizes the transitions incurred by this model. Most accesses are to objects in the exclusive states, for which same-state accesses do not require synchronization (*No sync.*). Few accesses trigger a conflicting transition (*Roundtrip coord.*), since the same object cannot trigger more than two conflicting transitions. However, a substantial fraction of accesses are to objects in shared states (*CAS*); each of these accesses requires an atomic operation.

	No sync. (excl. state)	**Roundtrip coord.** (confl. trans.)	**CAS** (shared state)
eclipse6	90.6%	0.00089%	9.4%
hsqldb6	94.3%	0.022%	5.6%
lusearch6	99.2%	0.00017%	0.98%
xalan6	87%	0.00033%	13%
avrora9	98.4%	0.0055%	1.6%
jython9	97.6%	0.0000021%	2.4%
luindex9	99.99982%	0.000098%	0.000062%
lusearch9	99.0%	0.000095%	1.0%
pmd9	97.0%	0.0023%	3.0%
sunflow9	42%	0.000056%	58%
xalan9	90.9%	0.00049%	9.1%
pjbb2000	86%	0.017%	14%
pjbb2005	90.6%	0.013%	9.3%

Table 4. State transitions for our implementation of von Praun and Gross's model. Rounding follows Table 3.

We find that this model slows programs by 2.2–3.1X (geomean is 2.2X excluding sunflow9 or 3.1X including it). For sunflow9, the slowdown is 221X (±9X) since it performs many accesses for which the instrumentation requires a CAS. Results for pessimistic barriers also show that these accesses are especially expensive for sunflow9 due to adding synchronized writes to mostly read-shared accesses. The other slowdowns vary significantly across programs: 17X for xalan9, 8.1X for xalan6, 1.3X for pjbb2005 (similar to the result for pessimistic barriers), and 1.3–2.2X for the rest. These results suggest that, when capturing cross-thread dependences, OCTET's design, which provides greater flexibility for handling shared access patterns, provides a significant advantage over von Praun and Gross's model.

6. Developing New Analyses

This section describes how OCTET can be used as a building block for new analyses. As described in preceding sections, OCTET guards all potentially shared accesses with barriers that coordinate access to objects. In addition, OCTET provides *hooks*, which the next section describes in more detail, that allow analyses to perform analysis-specific behavior when the barriers perform various actions. These features allow OCTET to provide two key capabilities that analyses can leverage:

- *Lightweight locks.* Some analyses, such as software transactional memory (STM) for enforcing atomicity [20], essentially need locks on all shared objects. OCTET's thread locality states allow OCTET barriers to function effectively as lightweight, biased, read/write locks on every shared object. By implementing hooks into OCTET's state transitions, analyses can perturb the execution, e.g., by resolving conflicts in an STM or by controlling interleavings to provide deterministic execution.
- *Tracking dependences and detecting conflicts.* Many analyses rely on detecting when a cross-thread dependence exists or when two accesses conflict, and taking some action in response. Analyses can implement "slow-path

	Lightweight locks	Track dependences	Precise accesses
Record deps.		✓	
Deterministic exec.	✓		
Check DRF/SC		✓	✓
Check atomicity		✓	✓
Enforce atomicity	✓	✓	✓*

Table 5. The analyses from Section 2.1 have different requirements. *When enforcing atomicity, last-access information such as read/write sets need not be fully precise.

hooks" (hooks that execute when OCTET performs a state transition) to identify all cross-thread dependences. The next section describes how a dependence recorder can build on OCTET to capture all cross-thread dependences.

Table 5 presents the five analyses from Section 2.1. The *Lightweight locks* and *Track dependences* columns show which analyses require these features. In general, "checking" and "recording" analyses need to track dependences, while "enforcing" analyses need lightweight locks. STM implementations probably need both—for conflict detection and resolution, respectively.

Our preliminary experience building on top of OCTET to enforce and check atomicity informs our understanding of these features, and suggests that Section 7's hooks are generally useful [7, 49, 56].

Developing precise analyses

Some analyses—in particular, "checking" analyses—require precision in order to avoid reporting false positives. While hooking into OCTET's transitions allows tracking dependences soundly, these dependences are not precise. The rest of this section identifies sources of precision that analyses require. A common theme is that providing precision is largely a thread-local activity, so analyses may be able to provide precision with reasonable overhead.

Prior accesses. Many checking analyses need to know not only that a cross-thread dependence exists, but also *when* the *source* of the dependence occurred. For example, when T2 reads o, an atomicity checker needs to know not only that T1 last wrote o but also "when" T1 wrote o (e.g., T1's atomic block or non-atomic access that wrote o) to identify the exact dependence and ultimately decide whether an atomicity violation exists. Analyses can provide this precision by storing thread-local *read and write sets* [38]. For example, atomicity checking would maintain read and write sets for each executed region.

Inferring precise dependences from imprecise transitions. To be sound, an analysis must assume the existence of cross-thread dependences implied by OCTET's happens-before edges. However, many of these dependences may not exist. A transition from RdSh to $\mathsf{WrEx}_{\mathsf{T}}$ implies a potential dependence from all other threads to T, but this dependence may not exist for every thread. In Figure 5, to capture the write–read dependence from T1 to T5, an analysis must capture all of the happens-before edges that imply this dependence. However, other implied dependences do not actually exist, e.g., no dependence exists from T1's write of o to the reads of p by T5 and T6.

Field granularity. Furthermore, all happens-before edges imply potentially imprecise dependences because of granularity mismatch: OCTET tracks state at object granularity for performance reasons, but precise analyses typically need field (and array element) granularity. While an analysis could modify OCTET to track dependences at field granularity to increase precision, the other forms of imprecision would still require mechanisms such as read/write sets to provide full precision.

The *Precise accesses* column of Table 5 identifies analyses that require precision. Detecting data races, SC violations, or atomicity violations requires perfect precision to avoid reporting false positives. Although enforcing atomicity with STM does not require full precision, since false positives affect performance but not correctness, this analysis would benefit from some precision about prior accesses, to avoid aborting transactions on every OCTET conflicting transition.

A precise analysis may be able to harness OCTET's imprecise happens-before edges as a sound, efficient "first-pass filter." Transitions that indicate a potential dependence would require a second, precise pass using precise thread-local accesses. If an analysis could avoid invoking the second pass frequently, it could achieve high performance by avoiding most synchronization in the common case.

7. Dependence Recorder

This section describes how we implement an analysis that records all of OCTET's happens-before edges, which soundly imply all cross-thread dependences. We describe "hooks" that OCTET provides for building new analyses and show how the dependence recorder implements these hooks.

Framework for soundly capturing dependences. In general, every analysis has some notion of what we call "dynamic access location" (DAL). A DAL could be defined as a dynamically executed region or transaction, in the case of an atomicity checker or STM. In the case of a dependence recorder, a DAL is a static program location plus per-thread dynamic counter, allowing an analysis to record *when* dependences occurred with enough granularity to support replay. The dependence recorder needs to capture both the *source* and *sink* DAL of each happens-before edge that OCTET establishes. The sink DAL is always the current memory access triggering an OCTET state transition; figuring out the source DAL is more challenging.

OCTET provides a hook for each state transition in Table 1 that analyses can implement in order to perform custom behavior when happens-before edges are established:

- handleConflictingTransitionAsRespondingThread(obj, oldState, accessType): During the explicit protocol, it is the *responding* thread that invokes this hook since the requesting thread is safely paused while the responding thread is active. The source DAL for the happens-before relationship is whichever DAL the responding thread is currently at. In Figure 4(a), the source DAL is point #4.
- handleConflictingTransitionAsRequestingThread(obj, oldState, accessType): Called during the implicit coordination protocol. The requesting thread must do any work to handle the conflicting transition, as the responding thread is blocked. The dependence recorder uses the safe point at which the responding thread is blocked as the source DAL of the happens-before edge. In Figure 4(b), the source DAL is point #2.
- handleUpgradingTransitionToRdSh(obj, oldState, accessType):[13] When transitioning obj from $\mathsf{RdEx}_{\mathsf{T1}}$ to $\mathsf{RdSh}_{\mathsf{c}}$, T2 establishes a happens-before relationship with T1. Because the DAL of T1's read may be hard to capture efficiently, the recorder captures the DAL from T1's most recent transition of *any* object to $\mathsf{RdEx}_{\mathsf{T1}}$, transitively capturing the necessary happens-before. The upgrading transition also establishes a happens-before edge from the last DAL to transition an object to RdSh (i.e., to $\mathsf{RdSh}_{\mathsf{c-1}}$). The recorder captures this DAL by having every upgrading transition to RdSh assign its DAL to a global variable, gLastRdSh, the most recent DAL to transition to RdSh.
- handleFenceTransition(obj, state, accessType): The dependence recorder uses the DAL recorded in gLastRdSh to capture the needed happens-before relationship.
- extendFastPath(obj, state, accessType): Tracking dependences does not require performing actions at the fast path, but other analyses can use this hook, e.g., to update local read/write sets.

Implementation and evaluation. We implement the dependence recorder on top of OCTET by implementing the hooks as described above. As each thread executes, it records happens-before edges in a per-thread log file. In addition to adding OCTET's instrumentation, the compilers add instrumentation at method entries and exits and loop back edges to update a per-thread dynamic counter (to help compute DAL), and each safe point records the current static program location, in case the safe point calls into the VM and responds to a coordination protocol request.

Figure 8 shows the overhead that the recorder adds on 32 cores. OCTET's overhead is the same as shown in Figure 6. We find it adds 5% on average over OCTET for an overall overhead of 31% (all percentages relative to baseline execution). By running subconfigurations of the recorder (not shown), we find that about half of the recorder's overhead over OCTET comes from the instrumentation added to track program location, and about half comes from actually recording OCTET's happens-before edges.

[13] Upgrading transitions from $\mathsf{RdEx}_{\mathsf{T}}$ to $\mathsf{WrEx}_{\mathsf{T}}$ capture a potential *intra*-thread dependence, so the dependence recorder, and most other analyses, do not need to hook onto it.

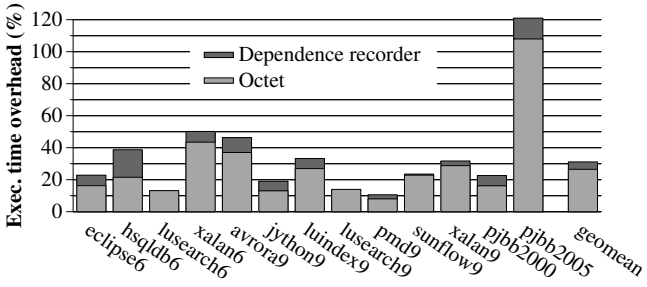


Figure 8. Run-time overhead that the dependence recorder adds on top of OCTET and overall.

Unsurprisingly, the recorder adds significant overhead to pjbb2005, which has the highest fraction of accesses triggering an OCTET state change. The recorder adds the most overhead (over OCTET) to hsqldb6, which performs a nontrivial number of RdSh→WrEx transitions (Table 3)—which involve as many as 100 responding threads (Table 2). OCTET performs these transitions inexpensively since most coordination uses the implicit protocol (Section 5.3), but recording a RdSh→WrEx transition incurs the cost of writing to each responding thread's log.

These results suggest that we can use OCTET to build a low-overhead dependence-recording analysis, potentially enabling online multithreaded record & replay.

8. Related Work

This section compares OCTET's concurrency control mechanism with prior work. While prior work employs optimistic synchronization for tracking ownership of shared memory, OCTET's design and purpose differ from prior approaches.

Biased locking. Prior work proposes *biased locking* as an optimistic mechanism for performing lock acquires without atomic operations [11, 25, 43, 46]. Each lock is "biased" toward a particular thread that may 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, and it introduces WrEx, RdEx, and RdSh states in order to support different sharing patterns efficiently. Hindman and Grossman present an approach similar to biased locking for tracking reads and writes in STM [22]. As with biased locking, their approach does not efficiently handle read-shared access patterns.

Cache coherence. OCTET's states correspond to cache coherence protocol states; its conflicting state transitions correspond to remote invalidations (Section 3.1). Thus, program

behavior that leads to expensive OCTET behavior may already have poor performance due to remote cache misses.

Cache coherence has been implemented in *software* in distributed shared memory (DSM) systems to reduce coherence traffic [4, 26, 27, 33, 47, 48]. *Shasta* and *Blizzard-S* both tag shared memory blocks with coherence states that are checked by barriers at each access [47, 48]. A coherence miss triggers a software coherence request; processors periodically poll for such requests.

While each unit of shared memory can have different states in different caches, each unit of shared memory has one OCTET state at a time. While cache coherence provides data consistency, OCTET provides concurrency control for analyses that need to capture cross-thread dependences.

Identifying shared memory accesses. Static approaches can identify definitely non-shared objects or non-racy accesses [13, 15, 52]. These approaches could complement OCTET by eliminating unnecessary barriers. *Chimera* lowers the cost of tracking cross-thread dependences by using static race detection, but it relies on profiling for efficiency [31].

Aikido avoids instrumenting accesses to non-shared memory by using OS paging and binary writing to limit instrumentation to reads and writes that might access shared memory [42]. As the remaining shared accesses still incur significant overhead, Aikido is complementary to OCTET.

Von Praun and Gross track thread ownership dynamically in order to detect races [51]. Section 5.5 compared our implementation of their state model with OCTET.

9. Summary

OCTET is a novel concurrency control mechanism that captures cross-thread dependences, without requiring synchronization at non-conflicting accesses. We have designed a state-based protocol, proven soundness and liveness guarantees, and described a framework for designing efficient analyses and systems on top of OCTET. An evaluation of our prototype implementation shows that real programs benefit from OCTET's optimistic tradeoff, and OCTET achieves overheads substantially lower than prior approaches.

A. OCTET Correctness Proofs

A.1 Proof of Theorem 1

Recall that we assume both fair thread scheduling and that all operations on queues will succeed. We begin by showing the following:

Lemma 1. *A thread will always eventually respond to* OCTET *requests from other threads.*

Proof. A thread has two means of responding to OCTET requests. A thread can *explicitly* respond to requests at safe points, and it will *implicitly* respond to requests as described in Section 3.3 if it is blocked. Hence, as long as non-blocked threads eventually block or reach a safe point, all requests will be responded to. Fair scheduling means that non-blocked threads make forward progress. Hence, it suffices to ensure that safe points are placed so that a thread cannot execute indefinitely without encountering one. As discussed in Section 3.3, safe points occur at least at loop back edges and method entries, and within all loops of the OCTET protocol outlined in Figure 3, ensuring that a non-blocked thread will eventually block or reach a safe point. □

The preceding Lemma readily yields the following:

Lemma 2. *A thread that changes the* OCTET *state of an object* o *will eventually be able to access* o*.*

Proof. If thread T changes o's state to any other state that does not require roundtrip coordination (i.e., T performs a $\mathsf{RdEx} \rightarrow \mathsf{RdSh}$ or $\mathsf{RdEx_T} \rightarrow \mathsf{WrEx_T}$ transition), then the access can proceed immediately. If T places an object in an intermediate state, then T cannot proceed until it receives responses from the necessary threads (a single thread in the case of a transition from WrEx or RdEx, or all threads in the case of a transition from RdSh). Lemma 1 says that all necessary responses will eventually arrive, and hence T can remove the intermediate flag and proceed with its access. □

We can now prove the following:

Theorem 1. OCTET*'s protocol is deadlock and livelock free.*

Proof. We note that showing deadlock and livelock freedom requires that at least one thread make progress when encountering an OCTET barrier. We can thus show that a thread at an OCTET barrier will either (a) successfully pass the barrier and complete its access; or (b) retry the barrier because a second thread has completed or will complete its access.

We thus consider a thread T attempting to access an object o, and consider each possibility under which the thread may attempt its access. These cases are labeled using tuples of the form (S, a), where S is the state o is in when T arrives at the OCTET barrier, and a denotes whether T wants to perform a read (r), a write (w), or either (r/w).

- $(\mathsf{WrEx_T}, r/w), (\mathsf{RdEx_T}, r)$**:** These are the simple cases. The OCTET barrier takes the fast path and T immediately proceeds to its access.
- (**Any intermediate state**, r/w)**:** If T finds o in an intermediate state, the OCTET protocol causes T to loop. However, in this situation, a second thread, $\mathsf{T'}$, has put o into an intermediate state, and, by Lemma 2, will eventually complete its access.
- $(\mathsf{WrEx_{T'}}, r/w), (\mathsf{RdEx_{T'}}, w), (\mathsf{RdSh_c}, w)$**:** In each of these cases, the conflicting transition protocol causes T to attempt to CAS o to the appropriate intermediate state. If the CAS fails, then some other thread $\mathsf{T'}$ put o into a different state, and, by Lemma 2, will make forward

progress. If the CAS succeeds, then T makes forward progress, instead.

($\mathbf{RdSh}_c, r$)**:** If necessary, T can update T.rdShCount without blocking. T then proceeds to its access.

($\mathbf{RdEx}_{T'}, r$)**:** T attempts to atomically increment gRdShCount. If the increment succeeds, T then attempts to CAS o's state to RdSh_c. If the CAS succeeds, T proceeds with its access, and if it fails, then some other thread T' performed a state change and is making forward progress by Lemma 2. If the atomic increment fails, then some thread T' is attempting the same transition, but in the "successful increment" case, and thus some thread is making forward progress.

($\mathbf{RdEx}_T, w$)**:** T attempts to upgrade o's state with a CAS. If the CAS succeeds, T proceeds with its access. If it fails, then some other thread changed o's state, and by Lemma 2 will complete its access.

Hence, in all cases, either T will eventually be able to proceed past the OCTET barrier and perform its access, or some other thread will successfully complete its access, and no deadlock or livelock is possible. □

A.2 Proof of Theorem 2

We next 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 compiler and hardware. We note that any cross-thread dependence only involves a single object. Dependences that involve two objects (e.g., loading a value from a field of one object and storing it into the field of a second) must happen within a single thread. We also assume that OCTET's instrumentation correctly ensures that an object is in a valid state before a thread performs its access (e.g., for T to write obj, obj must be in state WrEx_T).

Notation. We denote a read by thread T as r_T, and a write by T as w_T. 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 denote the object over which a dependence is carried as obj.

We will also use special notation for certain actions performed by threads when interacting with OCTET. $S\downarrow_T$ means that thread T put an object into OCTET state S. recv_T means T received a request on its request queue; resp_T means T responded; block_T means T has blocked its request queue; and $\mathsf{unblock}_T$ means T unblocked its request queue.

Theorem 2. 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:

$w_{T1} \rightarrow w_{T2}$**:** OCTET's barriers enforce that when T1 writes obj, obj must be in WrEx_{T1} state. When T2 attempts to perform its write, it will still find obj in WrEx_{T1} (because the dependence is not transitively implied, no other conflicting access to obj could have happened in the interim). T2 will put obj into $\mathsf{WrEx}^{\mathsf{Int}}_{T2}$ and make a request to T1.

In the case of the explicit response protocol, when T1 receives the request, it establishes $\mathsf{WrEx}^{\mathsf{Int}}_{T2}\downarrow_{T2}\rightarrow_{hb} \mathsf{resp}_{T1}$ (transitively implied by edge hb_1 in Figure 4(a)) and ensures that T1 will now see obj in state $\mathsf{WrEx}^{\mathsf{Int}}_{T2}$ (preventing future reads and writes by T1 to obj). When T2 sees the update of T1's response, it issues a fence, moves obj to state WrEx_{T2}, and proceeds with its write, establishing $\mathsf{recv}_{T1} \rightarrow_{hb} w_{T2}$ (transitively implied by edge hb_2 in Figure 4(a)) and hence $w_{T1} \rightarrow_{hb} w_{T2}$.

In the implicit response protocol, T2 moves obj to WrEx_{T2} only after observing that T1 is blocked. We thus have $\mathsf{WrEx}^{\mathsf{Int}}_{T2}\downarrow_{T2}\rightarrow_{hb} \mathsf{unblock}_{T1}$ (transitively implied by edge hb_1 in Figure 4(b)), ensuring that subsequent accesses by T1 happen after obj is moved to $\mathsf{WrEx}^{\mathsf{Int}}_{T2}$, and $\mathsf{block}_{T1} \rightarrow_{hb} w_{T2}$ (transitively implied by edge hb_2 in Figure 4(b)). Since $w_{T1} \rightarrow_{hb} \mathsf{block}_{T1}$, $w_{T1} \rightarrow_{hb} w_{T2}$ holds transitively.

$r_{T1} \rightarrow w_{T2}$**:** This dependence has two cases to handle.

Case 1: T2 finds the object in an exclusive state (either RdEx_{T1} or WrEx_{T1}). $r_{T1} \rightarrow_{hb} w_{T2}$ is established by the same roundtrip mechanism as in the prior scenario.

Case 2: T2 finds the object in RdSh state. In this case, the protocol for dealing with RdSh objects requires that T2 perform roundtrip coordination with *all* threads, establishing $r_{T1} \rightarrow_{hb} w_{T2}$.

$w_{T1} \rightarrow r_{T2}$**:** For thread T1 to write to obj, the object must be in WrEx_{T1} state. There are then three scenarios by which this dependence could occur.

Case 1: T2 is the first thread to read obj after the write by T1, so it will find obj in WrEx_{T1} state. This triggers roundtrip coordination and establishes $w_{T1} \rightarrow_{hb} r_{T2}$.

Case 2: T2 is the second thread to read obj after the write by T1. This means that there was some thread T3 that left the object in state RdEx_{T3}. By the previous case, we know $w_{T1} \rightarrow_{hb} \mathsf{RdEx}_{T3}\downarrow_{T3}$, with a fence between resp_{T1} (or block_{T1} in the case of the implicit protocol) and $\mathsf{RdEx}_{T3}\downarrow_{T3}$. Hence, when T2 uses a CAS to move the object to state RdSh, it establishes $\mathsf{resp}_{T1} \rightarrow_{hb} \mathsf{RdSh}\downarrow_{T2}$ (or $\mathsf{block}_{T1} \rightarrow_{hb} \mathsf{RdSh}\downarrow_{T2}$ in the case of the implicit protocol), enforcing $w_{T1} \rightarrow_{hb} r_{T2}$ transitively.

Case 3: T2 finds obj in RdSh_c state upon reading it. Note that by the previous case, there must be some thread T3 that placed obj in RdSh_c (establishing $w_{T1} \rightarrow_{hb} \mathsf{RdSh}_c\downarrow_{T3}$). To access obj in RdSh_c state, T2 checks T2.rdShCount $\geq$ c and, if the check fails, updates

T2.rdShCount with a fence to ensure that T2 last saw the value of gRdShCount *no earlier* than when T3 put obj in RdSh_c. Hence, we have $\mathsf{RdSh}_c \downarrow_{T3} \rightarrow_{hb} r_{T2}$, establishing $w_{T1} \rightarrow_{hb} r_{T2}$ transitively.

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

B. Eliminating Redundant Barriers

Not all OCTET barriers are necessary. A particular barrier may be "redundant" because a prior barrier for the same object guarantees that the object will have an OCTET state that does not need to change. The key insight in eliminating redundant barriers is that a thread can only "lose" access to an object when it reaches a safe point. Thus, an access does not need a barrier if it is always preceded by an access that guarantees it will have the right state, without any intervening operations that might allow the state to change. The following sections describe a *redundant barrier analysis* (RBA) and evaluate its effects on OCTET performance.

A barrier at an access A to object o is redundant if the following two conditions are satisfied along *every* control-flow path to the access:

- The path contains a prior access P to o that is at least as "strong" as A. Writes are stronger than reads, but reads are weaker than writes, so A's barrier is not redundant if A is a write and P is a read.
- The path does not execute a safe point between A and any last prior access P that is at least as strong as A.

We have designed a sound, flow-sensitive, intraprocedural data-flow analysis that propagates facts about accesses to all objects and statics, and merges facts conservatively at control-flow merges. The analysis is conservative about aliasing of object references, assuming they do not alias except when they definitely do. A potential safe point kills all facts, except for facts about newly allocated objects that have definitely not escaped.

Handling slow paths. Responding threads respond to requests explicitly or implicitly at safe points, allowing other threads to perform conflicting state changes on any object. Potential safe points include *every object access* because a thread may "lose" access to any object except the accessed object if it takes a slow path, which is a safe point. Thus, at an access to o, the *safe* form of our analysis kills data-flow facts for all objects except o.

We have also explored an *unsafe* variant of our analysis that does *not* kill data-flow facts at object accesses. This variant is interesting because some analyses built on OCTET could use it. Speculatively executed regions (e.g., using transactional memory [20]) can use the unsafe variant because "losing" any object triggers rollback.

	No RBA	Safe RBA (default)	Unsafe RBA
eclipse6	8.6×10^9	6.2×10^9	5.4×10^9
hsqldb6	3.7×10^8	3.3×10^8	3.2×10^8
lusearch6	1.6×10^9	1.2×10^9	1.1×10^9
xalan6	6.8×10^9	5.4×10^9	5.0×10^9
avrora9	4.0×10^9	3.1×10^9	2.5×10^9
jython9	3.5×10^9	2.7×10^9	2.5×10^9
luindex9	2.2×10^8	1.7×10^8	1.6×10^8
lusearch9	1.6×10^9	1.2×10^9	1.1×10^9
pmd9	4.0×10^8	3.1×10^8	2.8×10^8
sunflow9	1.2×10^{10}	8.7×10^9	5.3×10^9
xalan9	6.3×10^9	5.1×10^9	4.7×10^9
pjbb2000	1.1×10^9	9.2×10^8	8.4×10^8
pjbb2005	4.7×10^9	3.5×10^9	3.3×10^9

Table 6. Dynamic barriers executed under three RBA configurations: no analysis, safe analysis, and unsafe analysis.

Example. The following example code illustrates how barriers can be redundant:

```
o.f = ...

/* ... no loads or stores; no safe points; no defs of o ... */

// barrier not required
... = o.f;

// read barrier on p may execute slow path
... = p.g;

// barrier required by safe variant (not by unsafe variant)
... = o.f;
```

The first read barrier for o is unnecessary because o's state will definitely be WrEx_T. The second read barrier for o is necessary for the safe variant because the barrier for p may execute the slow path, which is a safe point.

Performance impact. All of the OCTET results presented in Section 5 eliminate redundant barriers based on the safe variant of RBA. This section evaluates the benefit of the analysis by comparing to configurations without RBA and with the unsafe variant of RBA.

Table 6 shows the number of dynamic barriers executed without and with RBA. *No RBA* is barriers inserted without RBA; *Safe RBA (default)* and *Unsafe RBA* are the barriers inserted after using RBA's safe and unsafe variants. We see that the safe variant of RBA is effective in reducing the number of OCTET barriers executed, and the unsafe variant usually eliminates a modest amount of additional barriers.

Evaluating performance with and without RBA, we find that safe RBA (the default) improves performance by 1% on average (relative to baseline execution time) compared with no RBA. Unsafe RBA improves performance on average by 1% over safe RBA. Unsafe RBA improves the performance of sunflow9 by 7% on average over safe RBA; sunflow9 is also the program for which unsafe RBA removes the largest fraction of barriers over safe RBA (Table 6).

While we focus here on objects that were previously accessed, other analyses could potentially identify objects that definitely cannot be *shared* and thus do not need barriers. Ultimately, these optimizations may be beneficial, but, as with

RBA, we expect them to have a muted effect on overall overhead, as non-shared objects will always take fast paths at OCTET barriers, and OCTET's fast-path overheads are low.

C. Implementing the Coordination Protocol

We have implemented the abstract protocol from Section 3.3 as follows. For its request queue, each thread maintains a linked list of requesting threads represented with a single word called req. This word combines three values using bitfields so that a single CAS can update them atomically:

counter: The number of requests made to this thread.

head: The head of a linked list of requesting threads.

isBlocked: Whether this thread is at a blocking safe point.

The linked list is connected via next pointers in the requesting threads. Because a requesting thread may be on multiple queues (if it is requesting access to a RdSh object), it has an array of next pointers: one for each responding thread. Each thread also maintains a counter resp that is the number of requests to which it has responded. A responding thread responds to requests by increasing its resp counter; in this way, it can respond to multiple requests simultaneously.

Figure 9 shows the concrete implementation of the abstract request queue using the req word and resp counter. Each request increments req.counter; it adds the requesting thread to the linked list if using the explicit protocol. Responding threads process each requesting thread in the linked list in an analysis-specific way (represented with the call to processList)—in reverse order if FIFO behavior is desired—and they respond by updating resp, which requesting threads observe.[14] The linked list allows responding threads to know *which* requesting thread(s) are making requests, which allows the responding thread to perform conflict detection based on a requesting thread's access, for example. On the other hand, analyses that only need to establish a happens-before relationship can elide all of the linked list behavior and use only the counters; in this case, the CAS loop in handleRequestsHelper must be replaced with a memory fence.

```
requestsSeen() { return this.req.counter > this.resp; }
handleRequests() {
  handleRequestsHelper(false);
}
handleRequestsAndBlock() {
  handleRequestsHelper(true);
}
handleRequestsHelper(isBlocked) {
  do {
    newReq = oldReq = this.req;
    newReq.isBlocked = isBlocked;
    newReq.head = null;
  } while (!CAS(&this.req, oldReq, newReq));
  processList(oldReq.head);
  this.resp = oldReq.counter;
}
resumeRequests() {
  do {
    newReq = oldReq = this.req;
    newReq.isBlocked = false;
  } while (!CAS(&this.req, oldReq, newReq));
}
```

(a) Methods called by responding thread respT, i.e., this is respT.

```
request(respT) {
  do {
    newReq = oldReq = respT.req;
    if (!oldReq.isBlocked) {
      this.next[respT] = oldReq.head;
      newReq.head = this;
    }
    newReq.counter = oldReq.counter + 1;
    if (CAS(&respT.req, oldReq, newReq))
      return oldReq.isBlocked;
  } while (true);
}
status(respT) { return respT.responses >= newReq.counter; }
```

(b) Methods called by requesting thread reqT, i.e., this is reqT. Note that status() uses the value of newReq from request().

Figure 9. Concrete implementation of request queues.

Acknowledgments

We thank Luis Ceze, Brian Demsky, Dan Grossman, Kathryn McKinley, Madan Musuvathi, and Michael Scott for valuable discussions and feedback; and the anonymous reviewers for helpful and detailed comments and suggestions.

References

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

[2] S. V. Adve and M. D. Hill. Weak Ordering—A New Definition. In *ISCA*, pages 2–14, 1990.

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

[5] L. Baugh, N. Neelakantam, and C. Zilles. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. In *ISCA*, pages 115–126, 2008.

[6] 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.

[7] S. Biswas, J. Huang, A. Sengupta, and M. D. Bond. DoubleChecker: Efficient Sound and Precise Atomicity Checking. Unpublished, 2013.

[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.

[14] If an analysis processes the requesting threads, the responding thread may need to issue a fence before updating resp, to ensure proper ordering.

[9] 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.

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

[11] M. Burrows. How to Implement Unnecessary Mutexes. In *Computer Systems Theory, Technology, and Applications*, pages 51–57. Springer–Verlag, 2004.

[12] L. Ceze, J. Devietti, B. Lucia, and S. Qadeer. A Case for System Support for Concurrency Exceptions. In *HotPar*, 2009.

[13] 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.

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

[15] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-Aware Java Runtime. In *PLDI*, pages 245–255, 2007.

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

[17] C. Flanagan and S. N. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *ACM Workshop on Program Analysis for Software Tools and Engineering*, pages 1–8, 2010.

[18] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *PLDI*, pages 293–303, 2008.

[19] K. Gharachorloo and P. B. Gibbons. Detecting Violations of Sequential Consistency. In *SPAA*, pages 316–326, 1991.

[20] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2nd edition, 2010.

[21] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, pages 289–300, 1993.

[22] B. Hindman and D. Grossman. Atomicity via Source-to-Source Translation. In *MSPC*, pages 82–91, 2006.

[23] 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.

[24] T. Kalibera, M. Mole, R. Jones, and J. Vitek. A Black-box Approach to Understanding Concurrency in DaCapo. In *OOPSLA*, pages 335–354, 2012.

[25] K. Kawachiya, A. Koseki, and T. Onodera. Lock Reservation: Java Locks Can Mostly Do Without Atomic Operations. In *OOPSLA*, pages 130–141, 2002.

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

[27] L. I. Kontothanassis and M. L. Scott. Software Cache Coherence for Large Scale Multiprocessors. In *HPCA*, pages 286–295, 1995.

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

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

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

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

[32] 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.

[33] 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.

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

[35] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *ISCA*, pages 210–221, 2010.

[36] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL*, pages 378–391, 2005.

[37] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *PLDI*, pages 351–362, 2010.

[38] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *HPCA*, pages 254–265, 2006.

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

[40] N. Nethercote and J. Seward. How to Shadow Every Byte of Memory Used by a Program. In *ACM/USENIX International Conference on Virtual Execution Environments*, pages 65–74, 2007.

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

[42] M. Olszewski, Q. Zhao, D. Koh, J. Ansel, and S. Amarasinghe. Aikido: Accelerating Shared Data Dynamic Analyses. In *ASPLOS*, pages 173–184, 2012.

[43] T. Onodera, K. Kawachiya, and A. Koseki. Lock Reservation for Java Reconsidered. In *ECOOP*, pages 559–583, 2004.

[44] M. S. Papamarcos and J. H. Patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *ISCA*, pages 348–354, 1984.

[45] 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.

[46] K. Russell and D. Detlefs. Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing. In *OOPSLA*, pages 263–272, 2006.

[47] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *ASPLOS*, pages 174–185, 1996.

[48] 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 *ASPLOS*, pages 297–306, 1994.

[49] A. Sengupta, S. Biswas, M. D. Bond, and M. Kulkarni. EnforSCer: Hybrid Static–Dynamic Analysis for End-to-End Sequential Consistency in Software. Technical Report OSU-CISRC-11/12-TR18, Computer Science & Engineering, Ohio State University, 2012.

[50] 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.

[51] C. von Praun and T. R. Gross. Object Race Detection. In *OOPSLA*, pages 70–82, 2001.

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

[53] L. Wang and S. D. Stoller. Runtime Analysis of Atomicity for Multithreaded Programs. *IEEE TSE*, 32:93–110, 2006.

[54] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking. Barriers Reconsidered, Friendlier Still! In *ISMM*, pages 37–48, 2012.

[55] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley. Why Nothing Matters: The Impact of Zeroing. In *OOPSLA*, pages 307–324, 2011.

[56] M. Zhang, J. Huang, and M. D. Bond. LarkTM: Efficient, Strongly Atomic Software Transactional Memory. Technical Report OSU-CISRC-11/12-TR17, Computer Science & Engineering, Ohio State University, 2012.