OCTET: Practical Concurrency Control for Dynamic Analyses and Systems

Michael D. Bond* Milind Kulkarni+ Meisam Fathi Salmi* Minjia Zhang*
Swarnendu Biswas* Jipeng Huang* Aritra Sengupta*

* Ohio State University
+ Purdue University

{mikebond,fathi,zhanminj,biswass,huangjip,sengupta}@cse.ohio-state.edu
milind@purdue.edu

Ohio State CSE technical report #OSU-CISRC-7/12-TR13, July 2012

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*—such as 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 cross-thread dependences (i.e., conflicting accesses) requires synchronization at every access to potentially shared memory.

This paper introduces a new software-based concurrency control mechanism called OCTET that captures cross-thread dependences soundly but avoids synchronization at non-conflicting accesses. OCTET tracks the *locality state* of each potentially shared object. Non-conflicting accesses conform to the locality state and require no synchronization, but conflicting accesses require a state change with heavyweight synchronization. This optimistic tradeoff performs well for real-world concurrent programs, which by design execute relatively few conflicting accesses.

We have implemented a prototype of OCTET in a high-performance Java virtual machine. Our evaluation demonstrates OCTET's potential for capturing cross-thread dependences with overhead low enough for production systems. OCTET is an appealing and practical concurrency control mechanism for designing low-overhead, sound and precise analyses and systems that check and enforce concurrency correctness properties.

1. Introduction

Software must become more concurrent to reap the benefits of hardware that provides *more*, instead of *faster*, cores with successive generations. However, parallel programming is notoriously difficult. A key challenge is that programmers must balance two competing concerns: *concurrency correctness*—atomicity, sequential consistency (SC), and determinism—and *performance*—single-thread overhead and multithreaded scalability.

Languages and systems can help by providing good support for guaranteeing concurrency correctness through sound enforcement or sound and precise checking, e.g., enforcing determinism through record & replay (e.g., DoublePlay [63]) and deterministic execution (e.g., CoreDet [7]); enforcing atomicity (transactional memory [26]); checking atomicity (e.g., Velodrome [24]); and checking SC and data race freedom (e.g., DRFx [45]).

This paper calls these approaches *dynamic analyses and systems for concurrency correctness* (DASCC; singular or plural). DASCC in production systems can provide concurrency correctness by *enforcing* properties, which improves programmability, reliability, and scalability by eliminating whole classes of errors; or by *checking* properties, which enables terminating executions that violate correctness, as well as detecting and diagnosing hard-to-reproduce errors that occur rarely and only in production.

Despite these benefits, modern languages and systems do not provide adequate support for concurrency correctness because all existing approaches are *impractical*: they either require custom hardware support, slow programs by about an order of magnitude, or have other serious limitations. We focus on the high overhead of software-based solutions. Across a variety of DASCC, the key cost is *capturing cross-thread dependences* (i.e., detecting conflicting accesses to shared memory). Software-based DASCC must conservatively assume that unless it can be proven otherwise, any access to potentially shared memory is involved in a data race. Thus, software-based DASCC require *synchronized* instrumentation in order to capture dependences soundly, often slowing programs by up to an order of magnitude.

Contributions

In this paper, we present OCTET, a novel dynamic concurrency control mechanism that captures cross-thread dependences by introducing instrumentation at all potentially shared reads and writes. However, unlike existing schemes, OCTET exploits *thread locality*: the notion that most accesses to memory—even shared memory—do not involve cross-thread dependences. OCTET's design allows it to detect efficiently (without synchronization) at run time when an access is *not* involved in a cross-thread dependence. As a result, expensive synchronization is only necessary when dependences might occur. *This approach leads to overhead primarily determined by the number of dependences, rather than the number of accesses.* Section 3 describes the design of OCTET, while Section 4 proves the soundness and liveness of the scheme.

We have implemented OCTET in a high-performance JVM (Section 5). To verify that OCTET is a suitable platform for DASCC, Section 6 evaluates the performance and behavior of our OCTET implementation on five 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 introduces relatively low overhead—significantly better than the overheads of prior mechanisms, and potentially low enough for production systems.

Because a variety of DASCC rely on taking action upon the detection of cross-thread dependences, OCTET can serve as a foundation for designing new, efficient DASCC. While this paper does *not* introduce new DASCC based on OCTET, Section 7 discusses potential opportunities and challenges for implementing efficient DASCC on top of OCTET.

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, sampling, custom hardware, and language support. Unfortunately, these techniques are unpractical for contemporary systems for a variety of reasons. Among others, sufficiently precise static analyses do not scale to large-scale systems; sampling approaches can miss concurrency bugs; hardware support does not exist in current architectures; and language-based approaches do not suffice for existing code bases. Section 8.2 discusses these approaches in more detail.

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

2.1 Dynamic Analyses and Systems for Concurrency Correctness (DASCC)

This section motivates and describes DASCC that guarantee concurrency correctness of three key properties—determinism, sequential consistency, and atomicity—by either (1) enforcing them soundly or (2) checking them soundly and precisely. More generally, researchers have recently stressed the importance of DASCC for providing programmability and reliability guarantees and simplifying overly-complex semantics [1, 15].

Multithreaded record & replay. Record & replay of multithreaded programs provides debugging and systems benefits. *Offline* replay allows programmers to reproduce production failures that occur rarely and only in production environments due to multithreaded nondeterminism. *Online* replay allows multiple machines to execute the same interleavings at the same time, enabling systems benefits such as replication-based fault tolerance.

Record & replay for uniprocessors is relatively straightforward: it is sufficient to record context switches and nondeterministic system events such as I/O and reading the system clock. Record & replay on multiprocessors is harder due to the frequency of potentially racy shared memory accesses, which require synchronized instrumentation to capture soundly [35]. *Chimera* rules out non-racy accesses using whole-program static analysis, which reports many false positives [36]. To achieve low overhead, Chimera relies on profiling runs to identify mostly non-overlapping accesses and expand synchronization regions.

Most high-performance multithreaded record & replay approaches sidestep the problem of capturing cross-thread dependences explicitly. Several support only online or offline replay but not both [37, 54, 68]. *DoublePlay* supports both online and offline replay but 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 [63].

Deterministic execution. An alternative to record & replay is executing multithreaded programs deterministically [7, 18, 19, 40, 50]. 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 [50], add high overhead [7, 18], require custom hardware [19], or provide per-thread address spaces and merge changes at synchronization points, which may not scale well to programs with fine-grained synchronization [40].

Guaranteeing sequential consistency. *Sequential consistency* (SC) is a strong memory model that is easy for programmers to reason about. An execution is SC if it is equivalent to some execution in which all operations are totally ordered and each thread's operations appear in program order [34]. A *data race* occurs if two conflicting accesses (at least one write) are not ordered by the *happens-before* relationship [33], i.e., they are not ordered by synchronization. Language memory models typically guarantee SC for data race–free (DRF) executions [1, 43],

Sound and precise checking of SC or DRF is expensive, even with recent innovations for happens-before race detectors [23, 25]. An attractive alternative to checking SC or DRF is the following relaxed constraints: DRF executions must report no violation, SC-violating executions must report a violation, and SC executions that have a data race may or may not report a violation [25]. Recent work applies this insight by checking for conflicts between overlapping, synchronization-free regions [42, 45, 61], but it relies on custom hardware support to detect conflicting accesses efficiently.

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 [22, 24, 67]. Notably, *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 and searching for cycles [24]. However, Velodrome cannot be used to check for atomicity violations in production software because it slows programs by about an order of magnitude.

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 [26]. Custom hardware-based approaches offer low overhead [46], but any real hardware support will likely be very limited [6],[2] making efficient software TM (STM) an important long-term goal. Existing STMs suffer from two major, related problems—poor performance and weak semantics—that have led researchers to question STM's real-world potential [14, 70]. 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, not just between transactions [60]. Achieving STM with high performance and strong semantics is a challenge because of the cost of capturing cross-thread dependences throughout program execution.

2.2 Capturing Cross-Thread Dependences

Despite its benefits, efficient software support for checking and enforcing concurrency correctness properties has remained elusive: existing sofware-based DASCC slow programs significantly, often by up to an order of magnitude. What makes this diverse collection of analyses and systems so slow? To capture concurrent behavior accurately, these DASCC must *capture cross-thread dependences*: two accesses executed by different threads that have any kind of data dependence: a true, anti, or output dependence. Figure 1 shows a potential cross-thread anti-dependence. Note that since this dependence only arises when `obj1` and `obj2` reference the same object,

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

[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 {
  last = obj.md; // load per−object metadata
} while (last == LOCKED ||
         !CAS(&obj.md, last, LOCKED}));
if (last != curr) {
  handlePotentialDependence(...);
}
obj.f = ...;   // program write
memfence;
obj.md = curr; // unlock and update metadata
```

Figure 2. Barrier that captures cross-thread dependences.

and when the threads interleave in a particular manner, it is hard to accurately predict this dependence statically (Section 8.2).

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., a data race), the barriers themselves 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, obj.md, to track the last thread to access the metadata. If the last thread changes, the barrier handles the potential cross-thread dependence in a DASCC-specific way. It creates an atomic region by locking the per-object metadata using an atomic compare-and-swap (CAS) instruction[4] while handling the potential dependence and then performing the program memory access.

We evaluate the overhead of such instrumentation, and find that adding a barrier to all potentially shared memory accesses slows programs by 4.0X on average, even without performing any DASCC-specific operations (i.e., handlePotentialDependence() is a no-op). Further, using a simpler barrier that performs a single CAS (but no fence) at each shared access still slows execution by 2.2X slower on average. These overheads are unsurprising because atomic instructions and memory fences serialize in-flight instructions, and atomic instructions invalidate cache lines. Section 6.4 describes these experiments in more detail.

We now sketch how two DASCC described in Section 2.1—record & replay and atomicity checking—rely on capturing cross-thread dependences.

Record & replay. A recorded execution must capture cross-thread dependences soundly so that a replayed execution can replay them faithfully. A recorded execution can use the barrier in Figure 2 at all accesses to potentially shared memory to ensure that the order of all dependent accesses, including racy accesses, is captured. It can record the dynamic dependences between threads using a notion of dynamic access location (e.g., static location plus a dynamic counter) and some kind of clocks such as Lamport or vector clocks [33]. A replayed execution re-executes these dependences by ensuring that any sink waits for the corresponding source to execute first.

[3] A read (or write) barrier is instrumentation that executes at every read (write) [9].

[4] The atomic instruction CAS(addr, oldVal, newVal) attempts to update addr from oldVal to newVal, returning true on success.

Checking atomicity. A sound and precise atomicity-checking analysis such as Velodrome checks conflict serializability (CS) by constructing the dynamic dependence graph and detecting cycles in it [24]. Each node in the graph is a dynamic atomic region or a unary (non-atomic) access. The analysis adds edges between consecutive nodes executed by the same thread, and between nodes that have a cross-thread dependence. A cycle in the graph soundly and precisely identifies a CS violation. To capture cross-thread dependences, this analysis adds significant overhead that is difficult to avoid. While other components add significant overhead, such as maintaining last-access information and performing cycle detection, these sources of overhead are amenable to significant optimizations that would be more worthwhile if the cost of capturing cross-thread dependences could be reduced significantly.

3. Capturing Cross-Thread Dependences Efficiently

This section describes our approach for efficiently and accurately detecting *cross-thread dependences* on shared objects: data dependences involving accesses to the same variable by different threads. In prior work, capturing cross-thread dependences in software has proven difficult. Because any potentially shared memory might be involved in a cross-thread dependence, prior approaches have used synchronization at essentially every read and write (Section 2).

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

To achieve this goal, we associate a *thread-locality state* with each potentially shared object, that describes which accesses will definitely not cause cross-thread dependences. These accesses proceed without synchronization, 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. Hence, our technique's synchronization costs are proportional to the number of *conflicting* shared memory accesses in a program, rather than all accesses or even shared accesses.

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

3.1 OCTET States

A thread-locality state for an object tracked by OCTET captures access permissions for an object: it specifies which accesses can be made to that object that definitely *do not* create any new cross-thread dependences. The possible OCTET states for an object are:

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

RdEx_T: Read exclusive for thread T. T may read (but not write) the object without synchronization.

RdSh_c: Read shared. Any thread T may read the object without synchronization, subject to an up-to-date thread-local counter: $T.\mathsf{rdShCount} \geq c$ (described shortly).

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 T.rdShCount $\geq$ 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 T.rdShCount $<$ c	(T.rdShCount = c)	Memory fence	Potentially yes
	Conflicting	$\mathsf{WrEx_{T1}}$	W by T2	$\mathsf{WrEx_{T2}}$	Roundtrip comm.	Potentially yes
		$\mathsf{WrEx_{T1}}$	R by T2	$\mathsf{RdEx_{T2}}$		
		$\mathsf{RdEx_{T1}}$	W by T2	$\mathsf{WrEx_{T2}}$		
		$\mathsf{RdSh_c}$	W by T	$\mathsf{WrEx_T}$		

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

Note the similarity of the thread-locality states to coherence states in the standard MESI cache-coherence protocol [53] (Section 8.1). Modified and 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 may access $\mathsf{WrEx_T}$ or $\mathsf{RdEx_T}$ objects.

3.2 High-Level Overview of State Transitions

OCTET inserts read and write barriers before every access of a potentially shared object, as shown in Figure 3. When a thread attempts to access an object, OCTET checks the state of that object. If the state permits the access, it proceeds without synchronization, and the overhead is merely the cost of the state check (the *fast path*). If the access is not permitted, OCTET initiates a coordination protocol to changes the object's state so that the access can be permitted (the *slow path*). Note that the object's state cannot be changed without synchronization, as other threads may simultaneously be trying to access the object. OCTET's protocols ensure that all other threads will see the state change before their next attempt to access the object. A state change without coordination might result in some cross-thread dependences being missed. (Imagine a cache coherence protocol that allowed a processor to upgrade a cache line from Shared to Exclusive without waiting for other processors to invalidate the line in their caches!)

Table 1 gives a high-level overview of 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 needs to be changed. Note that the type of coordination required to safely perform a state change differs based on the type of transition needed. Some transitions are *conflicting* transitions, as they require coordination (in particular, roundtrip communication) with other threads to resolve, as described in Section 3.3, while others are *upgrading* or *fence* transitions, as they do not require explicit coordination with other threads, but instead merely require synchronization (Section 3.4).

OCTET's coordination protocols ensure that a happens-before edge is inserted each time an object's state changes. While a particular state change may not directly imply a cross-thread dependence, the combined set of happens-before edges inserted by OCTET ensures that every cross-thread dependence is ordered by these happens-before relationships. A proof of OCTET's soundness is in Section 4, and Section 7 describes how DASCC can build on OCTET's happens-before edges 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 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 state change *requests* access by sending a message to the thread—called the *responding thread*—that currently has access. ($\mathsf{RdSh}\rightarrow\mathsf{WrEx}$ transitions involve multiple responding threads, but for simplicity we first consider only one responding thread.) When the responding thread is at a *safe point*, a point in the program that is definitely *not* between an OCTET barrier and its corresponding access, it *responds* to the requesting thread. Upon receiving the response, the requesting thread can change the object's state and proceed with its access. This roundtrip communication 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.

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, and so instead *implicitly* responds to any requests by setting a flag that the requesting thread can inspect. Safe points (both blocking and non-blocking) must occur frequently enough to ensure that threads will never find themselves unable to reach a safe point. Placing non-blocking safe points at loop back edges and method entries, and treating all blocking operations as blocking safe points, suffices.

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 T1 allows other (requesting) threads to signal that they desire access to objects "owned" by T1 (i.e., objects in $\mathsf{WrEx_{T1}}$, $\mathsf{RdEx_{T1}}$, or RdSh states). The queue is also used by T1 to indicate to any requesting threads that T1 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 T1:

requestsSeen() Returns true if there are any pending requests for objects owned by T1.

handleRequests() Handles and responds to pending requests.

```
if (obj.state != WrEx_T) {
  /* Slow path: change obj.state */
}
obj.f = ...;   // program write
```

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

Figure 3. OCTET instrumentation at a program write and read. T is the current thread.

At non-blocking safe points:

```
1 if (requestsSeen()) {
2   memfence;
3   handleRequests();
4 }
```

At blocking safe points:

```
5 handleRequestsAndBlock();
6 /* blocking actions */
7 resumeRequests();
```

(a) Responding thread (thread T1)

To move obj from WrEx_{T1} or RdEx_{T1} to WrEx_{T2}:

```
1  currState = obj.state; // WrEx_T1 or RdEx_T1 expected
2  while (currState == any intermediate state ||
3         !CAS(&obj.state, currState, WrEx^Int_T2)) {
4    // obj.state ← WrEx^Int_T2 failed
5    /* safe point: check for and respond to requests */
6    currState = obj.state;
7  }
8  response = request(getOwner(currState));
9  while (!response) {
10   /* safe point: check for and respond to requests */
11   response = status(getOwner(currState));
12 }
13 memfence; // ensure happens−before
14 obj.state = WrEx_T2;
15 /* proceed with access */
```

(b) Requesting thread (thread T2)

Figure 4. Protocol for conflicting state changes.

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

resumeRequests() Atomically unblocks the queue.

The next two methods are called by the requesting thread T2 and act on the queue of the responding thread T1:

request(T1) Makes a request to T1. Returns true if queue is blocked.

status(T1) Returns true if T2's request has been seen and handled by T1.

The protocol for handling conflicting accesses is shown in Figure 4.[5] Figure 5 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 two threads, responding thread T1 and requesting thread T2, where T1 has access to object obj in WrEx_{T1} or RdEx_{T1} state, and T2 wants to write it (reads work similarly).

Requesting thread. T2 first uses a compare-and-swap (CAS) to place obj into the desired final state with an *intermediate* flag set (line 3 in Figure 4(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. For brevity, we will henceforth 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 T2's CAS may fail, if a third thread simultaneously attempts to access obj. Before T2 retries its CAS (lines 2–7 in Figure 4(b)), it responds to any pending requests on *its* request queue, to avoid deadlock (line 5 in Figure 4(b)). After placing obj into an intermediate state, T2 adds a request to T1's request queue by invoking request(T1) (line 8 in Figure 4 (b)).

Responding thread. To *respond* to a request, T1 uses either the *explicit* response protocol or the *implicit* response protocol. Whenever T1 is at a non-blocking safe point, it can safely relinquish control of objects using the explicit protocol. T1 simply checks if there are any pending requests on its request queue and calls handleRequests to deal with them (lines 1–3 in Figure 4(a)). Because the protocol performs a fence before handleRequests, the explicit response protocol establishes two happens-before relationships: one between T2's initial request for access to obj and any subsequent attempt by T1 to access obj (at which point T1 will find obj in $\text{WrEx}^{\text{Int}}_{T2}$ state), and another between T1's response to T2 and T2's access to obj.

The implicit response protocol is used if respT is at a *blocking* safe point. Intuitively, if T2 knows that T1 is at a blocking safe point, it can proceed assuming T1 has responded. This is accomplished by having T1 atomically "block" its request queue, letting T2 see that T1 is at a blocking safe point (line 5 in Figure 4(a)). This protocol establishes a happens-before relationship from T2 to T1 as in the explicit protocol. Further, it establishes a relationship between T1's *entering the blocked state* and T2's access to obj.

The conflicting-access coordination protocol establishes happens-before relationships between *two* threads. Before a thread writes to a *RdSh* object, it performs the explicit or implicit coordination protocol with *every other* active thread.

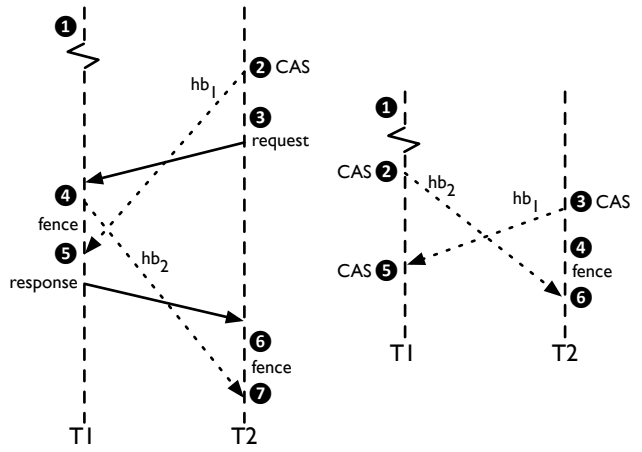
3.4 Handling Upgrading and Fence Transitions

A write by T1 to a RdEx_{T1} object triggers an *upgrading* transition to WrEx_{T1}, which requires an atomic instruction 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 6, 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 (RdSh→RdSh transitions, described next), OCTET globally orders all transitions to RdSh

[5] 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.



(a) Explicit protocol: (1) T1 writes to obj; (2) T2 places obj in $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T2}}$ with CAS (fence semantics); (3) T2 issues request to T1; (4) T1 receives request and issues fence; (5) T1 issues response, establishing hb_1; (6) T2 sees response and issues fence; (7) T2 places obj in $\mathsf{WrEx}_{\mathsf{T2}}$ and performs write, establishing hb_2.

(b) Implicit protocol: (1) T1 writes to obj; (2) T1 blocks with CAS (fence semantics); (3) T2 places obj in $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T2}}$ with CAS; (4) T2 observes T1 as blocked and issues a fence; (5) T1 unblocks with CAS, establishing hb_1; (6) T2 places obj in $\mathsf{WrEx}_{\mathsf{T2}}$ and performs write, establishing hb_2.

Figure 5. Operation of (a) explicit and (b) implicit coordination protocols for conflicting accesses to an object.

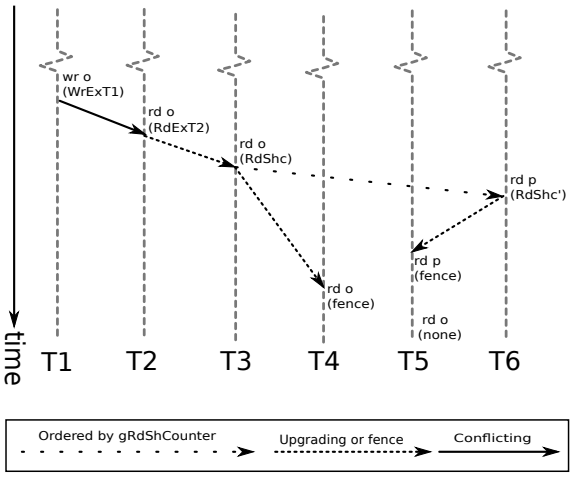


Figure 6. Example execution illustrating upgrading and fence transitions. After T1 writes o, it is in the $\mathsf{WrEx}_{\mathsf{T1}}$ state. Then T2 reads o, triggering a conflicting transition to $\mathsf{RdEx}_{\mathsf{T2}}$. Next, T3 reads o, triggering an upgrading transition to $\mathsf{RdSh}_{\mathsf{c}}$. T4 then reads o, triggering a fence transition, assuming it has *not* already read an object with state $\mathsf{RdSh}_{\mathsf{c'}}$ such that c'>c. T5 also reads o, but the read does *not* trigger a fence transition because T5 read an object p that we suppose transitioned to $\mathsf{RdSh}_{\mathsf{c'}}$ (by T6) more recently than o, i.e., c'>c.

states. To do so, it maintains a global counter gRdShCount that it increments atomically and uses the incremented value c in the new state $\mathsf{RdSh}_{\mathsf{c}}$.[6]

In Figure 6, T4 reads o in the $\mathsf{RdSh}_{\mathsf{c}}$ state. To capture the write–read dependence, T4 checks if its thread-local counter $\mathsf{T4.rdShCount} \geq \mathsf{c}$. If not, T4 triggers a *fence* transition because it has *not yet read o in the $\mathsf{RdSh}_{\mathsf{c}}$ state*. This transition issues a load fence to ensure a happens-before relationship with o's transition to $\mathsf{RdSh}_{\mathsf{c}}$ by T3, and updates $\mathsf{T4.rdShCount} = \mathsf{c}$.

T5's read of o does *not* trigger a fence transition because T5 previously read p in the $\mathsf{RdSh}_{\mathsf{c'}}$ state and c' > c. However, the write–read dependence is still 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. DASCC need all of these happens-before edges to capture all cross-thread dependences transitively (Section 7).

These mechanisms are required, ultimately, to ensure that in all cases OCTET enforces a happens-before relationship between the source and sink of all cross-thread dependences. The following section proves that OCTET's conflicting, upgrading, and fence transitions accomplish this goal, as well as proving that OCTET's transitions do not introduce deadlock or livelock.

4. Correctness of OCTET

This section demonstrates two critical correctness properties of OCTET. First, that 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 DASCC that might be built on top of OCTET, as described in Section 7. Second, that OCTET preserves liveness: despite the additional synchronization and control flow introduced by OCTET, no deadlock or livelock is introduced.

4.1 OCTET Soundness

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

We will assume, without loss of generality, that there is only one shared object, obj, and all cross-thread dependences arise through accesses to obj (interactions between multiple shared objects must happen within a single thread, and dependences between them will be preserved by the compiler's and hardware's reordering constraints). We also assume that OCTET's instrumentation behaves as expected, and hence OCTET ensures that an object is in a valid state before a thread performs its access (e.g., for thread T to write obj, the object must be in state $\mathsf{WrEx}_{\mathsf{T}}$).

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 also use special notation for certain actions performed by threads when interacting with OCTET. $\mathsf{S}\downarrow_{\mathsf{T}}$ means that thread T put an object into OCTET state S. $\mathsf{recv}_{\mathsf{T}}$ means T received a request on its request queue, and $\mathsf{resp}_{\mathsf{T}}$ means it responded; $\mathsf{block}_{\mathsf{T}}$ means T has blocked its request queue; and $\mathsf{unblock}_{\mathsf{T}}$ means T unblocked its request queue.

[6] Though in our experience a single global counter has sufficed to order accesses, potential scalability issues could be mitigated by using a global array of counters instead, with objects mapped to particular elements.

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

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

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

In the case of the explicit response protocol, when T1 receives the request, it establishes $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T2}} \downarrow_{\mathsf{T2}} \rightarrow_{hb} \mathsf{resp}_{\mathsf{T1}}$ (edge hb_1 in Figure 5(a)) and ensures that T1 will now see obj in state $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T2}}$ (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 $\mathsf{WrEx}_{\mathsf{T2}}$ and proceeds with its write, establishing $\mathsf{recv}_{\mathsf{T1}} \rightarrow_{hb} w_{T2}$ (edge hb_2 in Figure 5(a)) and hence $w_{T1} \rightarrow_{hb} w_{T2}$.

In the implicit response protocol, T2 moves obj to $\mathsf{WrEx}_{\mathsf{T2}}$ only after observing that T1 is blocked. We thus have $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T2}} \downarrow_{\mathsf{T2}} \rightarrow_{hb} \mathsf{unblock}_{\mathsf{T1}}$ (edge hb_1 in Figure 5(b)), ensuring that subsequent accesses by T1 happen after obj is moved to $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T2}}$, and $\mathsf{block}_{\mathsf{T1}} \rightarrow_{hb} w_{T2}$ (edge hb_2 in Figure 5(b)). Since $w_{T1} \rightarrow_{hb} \mathsf{block}_{\mathsf{T1}}$, $w_{T1} \rightarrow_{hb} w_{T2}$ holds transitively.

$r_{T1} \rightarrow w_{T2}$**:** There are two cases to deal with for this dependence.
Case 1: T2 finds the object in an exclusive state (either $\mathsf{RdEx}_{\mathsf{T1}}$ or $\mathsf{WrEx}_{\mathsf{T1}}$). $r_{T1} \rightarrow_{hb} w_{T2}$ 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, described in Section 3.3, requires that T2 perform roundtrip communication 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 $\mathsf{WrEx}_{\mathsf{T1}}$ 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 $\mathsf{WrEx}_{\mathsf{T1}}$ state. This triggers roundtrip communication 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 $\mathsf{RdEx}_{\mathsf{T3}}$. By the previous case, we know $w_{T1} \rightarrow_{hb} \mathsf{RdEx}_{\mathsf{T3}} \downarrow_{\mathsf{T3}}$, with a fence between $\mathsf{resp}_{\mathsf{T1}}$ (or $\mathsf{block}_{\mathsf{T1}}$ in the case of the implicit protocol) and $\mathsf{RdEx}_{\mathsf{T3}} \downarrow_{\mathsf{T3}}$. Hence, when T2 uses a CAS to move the object to state RdSh, it establishes $\mathsf{resp}_{\mathsf{T1}} \rightarrow_{hb} \mathsf{RdSh} \downarrow_{\mathsf{T2}}$ (or $\mathsf{block}_{\mathsf{T1}} \rightarrow_{hb} \mathsf{RdSh} \downarrow_{\mathsf{T2}}$ in the case of the implicit protocol), enforcing $w_{T1} \rightarrow_{hb} r_{T2}$ transitively.
Case 3: T2 finds obj in $\mathsf{RdSh}_{\mathsf{c}}$ state upon reading it. Note that by the previous case, there must be some thread T3 that placed obj in $\mathsf{RdSh}_{\mathsf{c}}$ (establishing $w_{T1} \rightarrow_{hb} \mathsf{RdSh}_{\mathsf{c}} \downarrow_{\mathsf{T3}}$). To access obj in $\mathsf{RdSh}_{\mathsf{c}}$ state, T2 checks $\mathsf{T2.rdShCount} \geq \mathsf{c}$ 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 $\mathsf{RdSh}_{\mathsf{c}}$. Hence, we have $\mathsf{RdSh}_{\mathsf{c}} \downarrow_{\mathsf{T3}} \rightarrow_{hb} r_{T2}$, establishing $w_{T1} \rightarrow_{hb} r_{T2}$ transitively.

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

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

4.2 OCTET Liveness

This section argues that, under realistic assumptions, the OCTET protocol is both deadlock- and livelock-free. In other words, there will never be a scenario where all threads are stuck at OCTET barriers and are unable to continue; at least one thread will be able to complete its access. Hence, if the target application is deadlock- and livelock-free, adding OCTET instrumentation will not introduce either pathology.

The two assumptions we make are: (i) the thread scheduler is *fair*, and no thread can be descheduled indefinitely; and (ii) attempts to add requests to the request queue will eventually succeed. The first assumption is true for most systems, and the second is true in practice for most concurrent queue implementations, given the first assumption.[7] Under these assumptions, we can show:

Theorem 2. *The* OCTET *protocol is deadlock- and livelock-free.*

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

We note that with some modifications, OCTET is not only deadlock- and livelock-free, but also starvation-free; not only will at least one thread make progress at all times, but no thread will never make progress. In the baseline design of OCTET, if multiple threads contend to place an object into an intermediate state, a given thread may never successfully do so, and hence starve. We can associate queues with objects that allow threads to queue up to place an object into an intermediate state, with only the object at the head of the queue allowed to change the state of an object. This means that all threads that want to place an object into an intermediate state will eventually be able to do so. From this fact, it is straightforward to show that all threads can make progress.

5. Implementation

We have implemented a prototype of OCTET in Jikes RVM 3.1.1, a high-performance Java virtual machine [2, 3].[8] Although Jikes RVM was originally developed as a research platform, developers have improved its robustness and performance significantly in recent years, and Jikes RVM now provides performance within 15% of commercial JVMs.[9]

OCTET*'s instrumentation.* Jikes RVM uses two dynamic compilers to transform bytecode into native code. The *baseline* compiler compiles each method when it first executes, converting bytecode directly to native code. The *optimizing* compiler recompiles hot (frequently executed) methods at increasing levels of optimization [4]. We modify both compilers in order to add barriers (cf. Figure 3) at every operation that reads or writes an object field, array element, or static field.

The implementation adds 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.

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, and one word per static field by inserting 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 $\mathsf{WrEx}_{\mathsf{T}}$ and $\mathsf{RdEx}_{\mathsf{T}}$ as the address of a thread object T, using the lowest bit (which is available due to

[7] The second assumption can be enforced trivially by using wait-free queues, but such implementations have high overheads in practice.

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

[9] `http://dacapo.anu.edu.au/regression/perf/9.12-bach.html`

objects being word aligned) to distinguish WrEx_T from RdEx_T. To represent RdSh_c, the implementation simply uses c, using a number range that cannot overlap with thread addresses:

- WrEx_T: &T (&T $<$ 0xc0000000)
- RdEx_T: &T | 0x1
- RdSh_c: c (c $>$ 0xc0000000)

Jikes RVM already reserves a register that always points to the current thread object (thrReg), so checking whether a state is WrEx or RdEx is extremely fast. To check whether an object's state is RdSh_c *and* T.rdShCount is up-to-date with c, the barrier simply compares the state with c. Our 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 the state is also RdSh). The implementation thus adds the following check before writes:

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

And it adds the follow check before reads:

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

The implementation initializes the OCTET state of newly allocated objects and newly resolved static fields to the WrEx_T state, where T is the allocating/resolving thread. Since the Java Memory Model does not provide a happens-before edge between object allocation and another thread's use of the object [43], a fast path might see an uninitialized metadata word (guaranteed to be zero), triggering the slow path, which simply waits for the value to be initialized.

Static optimizations. To some extent, OCTET obviates the need for static optimizations that identify accesses that cannot conflict, because it adds low 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 a static analysis that identifies and removes redundant barriers at compile time. Appendix B describes this analysis in more detail, and evaluates the performance impact of applying the analysis.

In addition to eliminating redundant barriers, our implementation also does not instrument accesses to final (immutable) fields and to known immutable classes such as String. Besides these cases, OCTET adds instrumentation at every object and static access in the application and libraries.

Conflicting transition 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 simply by incrementing its response counter; in this way, it can respond to multiple requests simultaneously.

```
requestsSeen() { return this.req.counter > this.resp; }

handleRequests() {
  handleRequestsHelper(false);
}

handleRequestsAndBlock() {
  handleRequestsHelper(true);
}

handleRequestsHelper(isBlocked) { // helper method
  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 T1, i.e., "this" is T1.

```
request(T1) {
  do {
    newReq = oldReq = T1.req;
    if (!oldReq.isBlocked) {
      this.next[T1] = oldReq.head;
      newReq.head = this;
    }
    newReq.counter = oldReq.counter + 1;
    if (CAS(&T1.req, oldReq, newReq))
      return oldReq.isBlocked;
  } while (true);
}

status(T1) { return T1.responses >= newReq.counter; }
```

(b) Methods called by requesting thread T2, i.e., "this" is T2. Note that status() uses the value of newReq from request().

Figure 7. Concrete implementation of abstract request queues.

Figure 7 shows our 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 a DASCC-specific way—in reverse order if FIFO behavior is desired—and they respond by updating resp, which requesting threads observe.[10]

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, DASCC that only need to establish a happens-before relationship can elide all of the linked list behavior and use only the counters.

[10] Note that if a DASCC requires that the list of requesting threads be processed, the responding thread may need to issue a fence before updating resp, to ensure proper ordering.

6. Evaluation

6.1 Methodology

Benchmarks. In our experiments, our modified Jikes RVM executes the parallel DaCapo Benchmarks [8] (version 2006-MR2) and a fixed-workload version of SPEC JBB2000 called pseudojbb [62]. The following table shows the number of application threads that each program executes: total threads executed (Column 1) and maximum threads running at any time (Column 2). Some values are ranges due to run-to-run nondeterminism.

	Total threads	Max live threads
eclipse	16–17	11
hsqldb	402	73–101
lusearch	65	65
xalan	9	9
pseudojbb	37	9

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

Measuring performance. To account for run-to-run variability due to dynamic optimization guided by timer-based sampling, we execute 50 trials for each performance result and take the median (to reduce the impact of performance outliers due to system noise) and verify that 95% confidence intervals are small (within 1% of total execution time). We build a high-performance configuration of Jikes RVM (FastAdaptive) that optimizes the VM ahead-of-time and adaptively optimizes the application at run time. Our performance results naturally include the cost of dynamic compilation, including OCTET's additional compilation costs (identifying redundant barriers and inserting barriers). We use Jikes RVM's high-performance generational Immix collector [10] and let the GC choose heap sizes adaptively.

6.2 State Transitions

Table 2 shows the number of OCTET transitions for our benchmarks, including accesses that do not change the state (i.e., fast path only). We execute a profiling configuration of OCTET that adds instrumentation to count these transitions. The three groups of columns show increasing levels of synchronization that correspond to the transitions in Table 1. The table shows that the vast majority of accesses do not require synchronization. Lightweight fast-path instrumentation handles these transitions (Figure 3). *Upgrading and fence* transitions occur roughly as often as *Conflicting* transitions; the conflicting transitions are more of a concern because we expect them to be more expensive. Conflicting transitions range from fewer than 0.001% (eclipse) to about 0.1% (hsqldb and xalan) of all transitions. The relative infrequency of conflicting transitions provides supporting evidence for OCTET's optimistic approach.

6.3 Performance

Figure 8 presents the run-time overhead OCTET adds to program execution. The overheads are normalized to unmodified Jikes RVM running the application. Each bar represents the run-time overhead added by OCTET, with sub-bars representing subsets of functionality. *Metadata only* is the overhead (including GC overhead) of adding a word to each object's header and for each static field, and initializing it to WrEx upon allocation, and is about 2% on average.

The configuration *No comm* adds OCTET barriers, but does not perform the conflicting transition protocol, measuring only OCTET's fast-path overhead. (We still allow state transitions to occur in this configuration; disabling them actually *slows* execution, as many fast-path checks fail since objects are often in a conflicting state.) OCTET's fast path adds 17% overhead on average, and about 27% to xalan, which has a high density of reads and writes.

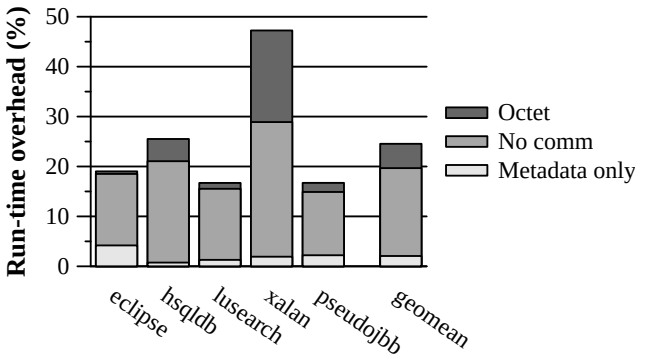


Figure 8. Components of OCTET run-time overhead.

Octet performs the conflicting transition protocol and adds 6% overhead on average. Overall OCTET overhead is 25% on average. Unsurprisingly, hsqldb and xalan, which perform a higher fraction of conflicting transitions (Table 2), add more communication overhead. This overhead is lower for hsqldb because many more of its conflicting transitions use the implicit protocol.

6.4 Comparison to Pessimistic Barriers

We have implemented and evaluated the "pessimistic" barrier in Figure 2 (Section 2.2) using the same implementation framework and experimental setup as for OCTET. The figure tracks only the last thread, but the implemented barriers use WrEx and RdSh states to handle read-shared access patterns. For simplicity of implementation, our barriers execute the program's write *outside* of the atomic region, an optimization that is acceptable for DASCC that capture only *unordered* conflicting accesses. We find that adding the barrier to all potentially shared memory accesses slows programs by 4.0X on average. We also evaluate a barrier that uses a single CAS but no additional memory fence, since some DASCC *may* be able to provide atomicity with such a barrier. This lower-bound configuration slows execution by 2.2X on average.

7. Discussion: Developing New DASCC

A key feature of OCTET's behavior is that its state transitions establish happens-before edges that transitively imply the execution's cross-thread dependences. DASCC can thus "piggyback" on these state transitions to capture all cross-thread dependences soundly. However, OCTET's happens-before edges over-approximate dependences, presenting challenges for DASCC that require precision.

7.1 Capturing Cross-Thread Dependences Soundly

Some DASCC, such as multithreaded record & replay, only need to capture cross-thread dependences soundly but not precisely. These DASCC can piggyback on each state transition to get the happens-before edge(s) for that transition. The transitive closure of these happens-before relationships over-approximates the execution's cross-thread dependences.

To handle happens-before edges, each DASCC has some notion of what we call "dynamic access location" (DAL). A DAL could be defined as a static program location plus per-thread dynamic counter in the case of record & replay; it could be a dynamically executed region or transaction in the case of an atomicity checker or STM. For any state transition, each happens-before edge's *sink* is the DAL of the memory access triggering the transition. The *source* depends on the type of state transition:

Conflicting transitions. Each conflicting transition naturally identifies happens-before edge(s) where each source is the DAL of a responding safe point. For example, in Figure 5(a) (page 6), source DAL is point #4; in Figure 5(b), it is point #2.

	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
eclipse	17,401,356,359 (**99.9987%**)				80,581 (**0.00046**%)			154,192 (**0.00089**%)			
	1.9%	89%	2.5%	6.7%	0.000086%	0.00038%	0.0000010%	0.0000073%	0.00071%	0.000012%	0.00016%
hsqldb	762,914,193 (**99.81**%)				567,794 (**0.074**%)			902,533 (**0.12**%)			
	3.3%	91.9%	0.44%	4.1%	0.048%	0.017%	0.0088%	0.038%	0.068%	0.00054%	0.011%
lusearch	3,101,841,194 (**99.99957**%)				5,383 (**0.00017** %)			7,926 (**0.00026**%)			
	1.8%	92.5%	4.25%	1.4%	0.00011%	0.000010%	0.000049%	0.0000068%	0.00024%	0%	0.0000056%
xalan	12,311,443,825 (**99.84**%)				7,992,383 (**0.065** %)			12,143,657 (**0.098**%)			
	0.94%	86%	0.13%	12%	0.065%	0.000021%	0.000035%	0.034%	0.065%	0.0000000081%	0.0000051%
pseudojbb	2,141,003,817 (**99.91**%)				910,542 (**0.042**%)			952,750 (**0.044**%)			
	2.8%	81%	1.4%	14%	0.038%	0.0047%	0.000079%	0.00026%	0.044%	0.0000021%	0.000071%

Table 2. OCTET state transitions, including fast-path executions that do not change the state. The first row for each benchmark is transitions for the column group, and the second row shows transitions of each type as a percentage of all transitions. We round percentages x as much as possible such that x and $100\% - x$ each have at least two significant digits.

Upgrading transitions. In Figure 6 (page 6), an upgrading transition from $\mathsf{RdEx}_{\mathsf{T2}}$ to $\mathsf{RdSh}_{\mathsf{c}}$ on T3 establishes a happens-before edge whose source is the DAL on T2 that transitioned o to $\mathsf{RdEx}_{\mathsf{T2}}$. This DAL probably cannot be obtained efficiently, but it is sufficient to capture the DAL from T2's most recent transition of *any* object to $\mathsf{RdEx}_{\mathsf{T2}}$, in order to establish a transitive happens-before relationship from the DAL that transitioned o to $\mathsf{RdEx}_{\mathsf{T2}}$.

Each upgrading transition to RdSh also establishes a happens-before edge from the last DAL to transition an object to RdSh (i.e., to $\mathsf{RdSh}_{\mathsf{c}-1}$). A DASCC can get this DAL by having every upgrading transition to RdSh assign its DAL to a global variable that is the most recent DAL to transition to RdSh.

Fence transitions. A fence transition establishes a happens-before edge whose source is the DAL that transitioned the object to RdSh, which is stored in a global variable as described above. A DASCC can thus handle this happens-before edge.

7.2 Developing Precise DASCC

While OCTET's happens-before edges capture cross-thread dependences soundly, it over-approximates the dependences, presenting challenges for developing precise DASCC.

Precise prior accesses. Many DASCC need to know precisely *when* the *source* of a cross-thread dependence occurred. For example, an atomicity checker needs to know which region last accessed an object. Even DASCC such as STM that do not require perfect precision, still need precise prior accesses to avoid detecting too many false conflicts.

DASCC can provide this precision by storing thread-local *read and write sets* [46]. For example, atomicity checking would maintain read and write sets for each executed region. Updates to read and write sets are thread local, so they can be relatively inexpensive, especially with lightweight static analysis and low-level optimizations such as non-temporal stores.

Imprecise transitions. It is nontrivial to determine the precise dependences from the happens-before edges that are based on OCTET's state transitions. In Figure 6, identifying the potential write–read dependences from T1 to T3 and T4 requires traversing happens-before edges transitively. T5's read of o does not even trigger a transition to indicate the write–read dependence T1 to T5, but rather it is transitively implied by happens-before edges that involve accesses to p and updates to gRdShCount. Furthermore, *all* happens-before edges imply potentially imprecise dependences because of granularity mismatch: OCTET tracks state at object granularity for performance reasons, but precise DASCC typically require field and array element granularity.

We believe that precise DASCC can harness OCTET's imprecise happens-before edges as a sound, effective "first-pass filter." Transitions that indicate a potential dependence require a second, precise pass that uses precise prior-access information such as the aforementioned thread-local read and write sets. If these DASCC can avoid invoking the second pass frequently, they can achieve high performance.

8. Related Work

8.1 Concurrency Control Mechanisms

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 purpose and design differ from prior approaches.

Biased locking. Prior work proposes *biased locking* as an optimistic mechanism for performing lock acquires without atomic operations [30, 52, 56], now implemented in major commercial Java virtual machines. Each lock is "biased" toward a particular thread that can acquire the lock without synchronization; other threads must communicate with the biasing thread before acquiring the lock. In contrast, OCTET applies an optimistic approach to all program accesses, not just locks, and it introduces WrEx, RdEx, and RdSh states in order to support different sharing patterns efficiently. OCTET's conflicting transition protocol is lightweight compared to biased locking's communication mechanisms, which is important since regular memory accesses occur more frequently than synchronization operations. Hindman and Grossman present an approach similar to biased locking for tracking reads and writes in STM [27]. As with biased locking, their approach does not handle read-shared access patterns efficiently.

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 [5, 31, 32, 39, 57, 58]. *Shasta* and *Blizzard-S* both tag shared memory blocks with coherence states, which instrumentation at each access checks [57, 58]. 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 DASCC that need to capture cross-thread dependences.

Identifying shared memory accesses. Olszewski et al. propose an approach called *Aikido* that avoids instrumenting accesses to

non-shared memory [51]. Aikido uses OS paging to identify shared memory and binary rewriting to add instrumentation only at reads and writes that might access shared memory. It still adds significant overhead to shared memory accesses, even if the accesses are mostly thread local; and page-level tracking identifies some unshared memory as shared.

Von Praun and Gross describe an approach for detecting races based on tracking thread ownership of objects [65]. Their ownership system dynamically identifies shared objects, allowing the race detector to restrict its attention to those shared 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; objects that enter shared states cannot return to an exclusive state. In contrast, OCTET supports transitioning back to exclusive states, and object accesses require synchronization only on state changes. OCTET's more precise tracking of ownership thus requires less synchronization.

8.2 Alternatives to Software-Based DASCC

This section discusses alternatives to dynamic, sound, software-based approaches.

Static approaches. Static program analysis (e.g., [48]), type checking (e.g., [13, 21]) model checking (e.g., [47]), and verification (e.g., [38]) can check concurrency correctness properties. These approaches are ahead-of-time and typically sound, so they can ensure a program is correct before running it. For example, conservative static analysis can determine that two object references cannot alias (point to the same object) or cannot execute at the same time when they alias [48]. However, static approaches do not currently scale well to large, complex programs, nor to dynamic language features such as dynamic class loading. Faced with uncertainty and complexity, static analysis reports false positives, which frustrate developers and make the analysis unusable if it reports many false positives; type checking similarly rejects correct programs; and full verification and model checking are unable to scale to large programs.

Static approaches can aid dynamic approaches by ruling out definitely non-shared objects or non-racy accesses. Prior work that uses static race detection to improve the performance of dynamic race detection shows that the fraction of accesses ruled out by static analysis varies significantly from program to program [17, 20, 66]. The resulting dynamic race detectors are still too expensive for production systems. Chimera uses static race detection to lower the cost of tracking cross-thread dependences, but it relies on profiling runs to be efficient [36].

Custom hardware support. Custom hardware support can capture cross-thread dependences efficiently by piggybacking on cache coherence protocols [6, 28, 41, 42, 46, 49, 61, 64, 69]. However, manufacturers have been slow to adopt such hardware support, which would add significant complexity to already-complex coherence protocols. If and when hardware supports capturing conflicting dependences, it is likely to be very limited, resulting in hybrid hardware–software approaches [6], so efficient software-based support is still required.

We believe that manufacturers will add such hardware support if programmer demand for it increases. Our work has the potential to make software-based DASCC fast enough for widespread use in production systems, ultimately leading programmers to demand even faster support in hardware.

Unsound approaches. Sampling-based analyses can find bugs probabilistically with low overhead [12, 29, 44]. However, sampling is (dynamically) unsound and cannot provide all-the-time checking and enforcement of concurrency correctness properties.

Prior work infers concurrency bugs by identifying concurrent program behavior that is well correlated with failure [29, 41, 59]. These approaches are neither sound nor precise but can be useful, especially for errors that violate unspecified properties such as atomicity. Existing approaches must capture cross-thread dependences, so they could benefit from our work.

Other programming models. Languages such as X10 help programmers specify parallelism better [16], although concurrency correctness remains an issue. Languages such as Deterministic Parallel Java (DPJ) and Jade provide determinism at the language level and thus avoid many concurrency bugs [11, 55], but programmers must rewrite their programs in these languages. Functional and declarative languages and domain-specific languages such as SQL and MATLAB avoid most or all concurrency bugs, but they do not seem to be able to solve many problems efficiently.

9. Summary

OCTET is a novel, optimistic concurrency control mechanism that captures cross-thread dependences, without requiring synchronization at non-conflicting accesses. We have designed a state-based protocol and proven soundness and liveness guarantees. 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 that need cross-thread dependences. We have described a general framework for designing sound and precise DASCC that will potentially have low enough overhead for all-the-time use in production systems.

Acknowledgments

Thanks to Luis Ceze, Brian Demsky, and Michael Scott for helpful discussions and feedback.

References

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

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

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

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

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

[6] L. Baugh, N. Neelakantam, and C. Zilles. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. In *ACM/IEEE International Symposium on Computer Architecture*, pages 115–126, 2008.

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

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

[9] S. M. Blackburn and A. L. Hosking. Barriers: Friend or Foe? In *ACM International Symposium on Memory Management*, pages 143–151, 2004.

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

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

[12] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional Detection of Data Races. In *ACM Conference on Programming Language Design and Implementation*, pages 255–268, 2010.

[13] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230, 2002.

[14] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *Communications of the ACM*, 51(11):40–46, 2008.

[15] L. Ceze, J. Devietti, B. Lucia, and S. Qadeer. A Case for System Support for Concurrency Exceptions. In *USENIX Conference on Hot Topics in Parallelism*, 2009.

[16] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005.

[17] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *ACM Conference on Programming Language Design and Implementation*, pages 258–269, 2002.

[18] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient Deterministic Multithreading through Schedule Relaxation. In *ACM Symposium on Operating Systems Principles*, pages 337–351, 2011.

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

[20] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-Aware Java Runtime. In *ACM Conference on Programming Language Design and Implementation*, pages 245–255, 2007.

[21] C. Flanagan and S. N. Freund. Type Inference Against Races. *Science of Computer Programming*, 64(1):140–165, 2007.

[22] C. Flanagan and S. N. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. *Science of Computer Programming*, 71(2):89–109, 2008.

[23] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *ACM Conference on Programming Language Design and Implementation*, pages 121–133, 2009.

[24] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *ACM Conference on Programming Language Design and Implementation*, pages 293–303, 2008.

[25] K. Gharachorloo and P. B. Gibbons. Detecting Violations of Sequential Consistency. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 316–326, 1991.

[26] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ACM/IEEE International Symposium on Computer Architecture*, pages 289–300, 1993.

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

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

[29] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and Sampling Strategies for Cooperative Concurrency Bug Isolation. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 241–255, 2010.

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

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

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

[33] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

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

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

[36] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: Hybrid Program Analysis for Determinism. In *ACM Conference on Programming Language Design and Implementation*, pages 463–474, 2012.

[37] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 77–90, 2010.

[38] K. R. Leino and P. Müller. A Basis for Verifying Multi-threaded Programs. In *European Symposium on Programming Languages and Systems*, pages 378–393, 2009.

[39] 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.

[40] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient Deterministic Multithreading. In *ACM Symposium on Operating Systems Principles*, pages 327–336, 2011.

[41] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

[42] 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 *ACM/IEEE International Symposium on Computer Architecture*, pages 210–221, 2010.

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

[44] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *ACM Conference on Programming Language Design and Implementation*, pages 134–143, 2009.

[45] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *ACM Conference on Programming Language Design and Implementation*, pages 351–362, 2010.

[46] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *International Symposium on High-Performance Computer Architecture*, pages 254–265, 2006.

[47] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *ACM Conference on Programming Language Design and Implementation*, pages 446–455, 2007.

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

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

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

[51] M. Olszewski, Q. Zhao, D. Koh, J. Ansel, and S. Amarasinghe. Aikido: Accelerating Shared Data Dynamic Analyses. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 173–184, 2012.

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

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

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

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

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

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

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

[59] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do I Use the Wrong Definition?: DefUse: Definition-Use Invariants for Detecting Concurrency and Sequential Bugs. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 160–174, 2010.

[60] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing Isolation and Ordering in STM. In *ACM Conference on Programming Language Design and Implementation*, pages 78–88, 2007.

[61] A. Singh, D. Marino, S. Narayanasamy, T. Millstein, and M. Musuvathi. Efficient Processor Support for DRFx, a Memory Model with Exceptions. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–66, 2011.

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

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

[64] E. Vlachos, M. L. Goodstein, M. A. Kozuch, S. Chen, B. Falsafi, P. B. Gibbons, and T. C. Mowry. ParaLog: Enabling and Accelerating Online Parallel Monitoring of Multithreaded Applications. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 271–284, 2010.

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

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

[67] L. Wang and S. D. Stoller. Runtime Analysis of Atomicity for Multithreaded Programs. *IEEE Transactions on Software Engineering*, 32:93–110, 2006.

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

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

[70] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 265–274, 2008.

A. Proof of Theorem 2

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 previously, safe points occur at least at loop back edges and method entries, and within all loops of the OCTET protocol outlined in Figure 4, ensuring that a non-blocked thread will always either 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 T changes o's state to any other state that does not require roundtrip communication (i.e., T performs a $\mathsf{RdEx}_{\mathsf{T}'} \rightarrow \mathsf{RdSh}$ or $\mathsf{RdEx}_{\mathsf{T}} \rightarrow \mathsf{WrEx}_{\mathsf{T}}$ transition), then the access can proceed immediately. If a thread 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 2. *The* OCTET *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).

- $(\mathsf{WrEx}_{\mathsf{T}}, r/w), (\mathsf{RdEx}_{\mathsf{T}}, r)$**:** 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, T', has put o into an intermediate state, and, by Lemma 2, will eventually complete its access.
- $(\mathsf{WrEx}_{\mathsf{T}'}, r/w), (\mathsf{RdEx}_{\mathsf{T}'}, w), (\mathsf{RdSh}_{\mathsf{c}}, w)$**:** 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 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.
- $(\mathsf{RdSh}_{\mathsf{c}}, r)$**:** If necessary, T can update T.rdShCount without blocking. T then proceeds to its access.
- $(\mathsf{RdEx}_{\mathsf{T}'}, r)$**:** T attempts to atomically increment gRdShCount. If the increment succeeds, T then attempts to CAS o's state to $\mathsf{RdSh}_{\mathsf{c}}$. 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.
- $(\mathsf{RdEx}_{\mathsf{T}}, w)$**:** 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. □

```
o.f = ...

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

// barrier unnecessary
... = o.f;

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

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

Figure 9. Example of redundant barriers.

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.

B.1 Redundant Barrier Analysis (RBA)

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 must 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 all 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 we could make it safe by modifying each barrier slow path so that it performs a read or write barrier on each object in the data-flow facts at that point. These additional barriers would need to complete without executing a slow path—otherwise an object's state might change—which could be accomplished by retrying them until none executed a slow path. We have not implemented this optimization in the slow path, but we do implement and evaluate the unsafe data-flow analysis in order to establish an upper bound on the potential performance benefit.

Example. Figure 9 shows an example of a redundant barrier and a barrier that is redundant for the unsafe but not the safe variant.

	No RBA	Safe RBA (default)	Unsafe RBA
eclipse	123,892	101,484 (82%)	100,188 (81%)
hsqldb	20,420	16,714 (82%)	15,933 (78%)
lusearch	22,526	17,640 (78%)	16,323 (72%)
xalan	36,875	30,111 (82%)	29,667 (80%)
pseudojbb	21,247	17,276 (81%)	16,431 (77%)

Table 3. Static barriers inserted under three RBA configurations: no analysis, safe analysis, and unsafe analysis.

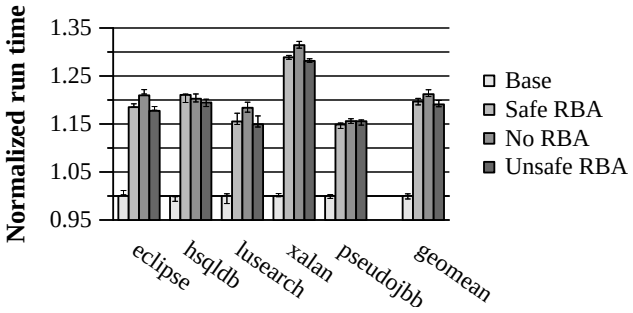


Figure 10. Run-time overhead of the OCTET fast path without and with redundant barrier analysis, and with the unsafe variant. The ranges are 95% confidence intervals, centered at the mean.

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.

B.2 Performance Impact of Redundant Barrier Analysis

All of the OCTET results presented in Section 6 eliminate redundant barriers based on the safe variant of RBA. This section evaluates the benefit of the analysis by comparing to a configuration without RBA. We also evaluate the potential for the slow path optimization described in Section B.1, by evaluating the unsafe variant of RBA.

Table 3 shows the number of (static) barriers inserted by the compiler without and with RBA. *No RBA* is barriers inserted without RBA; *Safe RBA (default)* and and *Unsafe RBA* are the barriers inserted after using RBA's safe and unsafe variants, respectively. We see that the safe variant of RBA is effective in reducing the number of barriers inserted by OCTET, and the unsafe variant removes a modest amount of additional barriers.

Figure 10 compares the performance of three RBA configurations. For simplicity we evaluate only the overhead of the fast path using the *No comm* configuration; *Safe RBA* is thus the same result as *No comm* in Figure 8. *No RBA* does *not* use RBA, slowing execution by 1–2% of base program execution time or 8% of *Safe RBA*'s overhead. The number of static barriers removed (18–22%) does not translate into a larger performance win. *Unsafe RBA* uses the unsafe variant of RBA analysis in order to evaluate the potential for the slow path optimization described in Section B.1. The unsafe variant indicates that the slow path optimization could improve performance by less than 1% of execution time or 3% of *Safe RBA*'s overhead.

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.