Drinking from Both Glasses: Combining Pessimistic and Optimistic Tracking of Cross-Thread Dependences *

Man Cao   Minjia Zhang   Aritra Sengupta   Michael D. Bond

Ohio State University (USA)

{caoma,zhanminj,sengupta,mikebond}@cse.ohio-state.edu

Abstract

It is notoriously challenging to develop parallel software systems that are both scalable and correct. Runtime support for parallelism—such as multithreaded record & replay, data race detectors, transactional memory, and enforcement of stronger memory models—helps achieve these goals, but existing commodity solutions slow programs substantially in order to track (i.e., detect or control) an execution's cross-thread dependences accurately. Prior work tracks cross-thread dependences either "pessimistically," slowing every program access, or "optimistically," allowing for lightweight instrumentation of most accesses but dramatically slowing accesses involved in cross-thread dependences.

This paper seeks to *hybridize* pessimistic and optimistic tracking, which is challenging because there exists a fundamental mismatch between pessimistic and optimistic tracking. We address this challenge based on insights about how dependence tracking and program synchronization interact, and introduce a novel approach called *hybrid tracking*. Hybrid tracking is suitable for building efficient runtime support, which we demonstrate by building hybrid-tracking-based versions of a dependence recorder and a region serializability enforcer. An adaptive, profile-based policy makes runtime decisions about switching between pessimistic and optimistic tracking. Our evaluation shows that hybrid tracking enables runtime support to overcome the performance limitations of both pessimistic and optimistic tracking alone.

1. Introduction

Software must become more parallel in order to scale with successive microprocessor generations that provide more, instead of faster, cores. However, writing and debugging parallel programs is notoriously difficult. General-purpose programming languages provide *shared memory and locks*, which are simple to understand, but hard to use to achieve both correctness and scalability.

Researchers have developed dynamic program analyses and software systems that help support reliable, scalable parallelism. This paper uses the general term "runtime support" to refer to such analyses and systems, which check or enforce concurrency correctness properties such as atomicity, determinism, and data race freedom. Notable examples of runtime support include data race detectors (e.g., [18]), software transactional memory (e.g., [20]), enforcement of strong memory models (e.g., [29]), atomicity checkers (e.g., [19]), and multithreaded record & replay (e.g., [38]). However, existing instances of runtime support are *impractical* because they slow programs substantially, rely on custom hardware, or have other serious limitations.

Existing runtime support for commodity systems (often called *software-only*) adds expensive instrumentation at each program access in order to *track* (detect or control) *cross-thread dependences* (data dependences involving two threads). This instrumentation is particularly costly because it must add its own synchronization in order to ensure soundness in the presence of data races in the program execution. Most existing runtime support uses an atomic operation at every access (e.g., [18–20, 24, 25]), which we refer to as *pessimistic tracking* of dependences. The performance of runtime support built on pessimistic tracking is relatively insensitive to the number of cross-thread dependences in an execution. However, its frequent synchronization typically slows executions by several times or more. Alternatively, *optimistic tracking* avoids synchronization for accesses *not* involved in cross-thread dependences, but requires coordination between threads when accesses *are* involved in dependences [11, 13, 22, 33, 35, 39]. We emphasize that although optimistic tracking performs well for the many programs that perform relatively few conflicting accesses, its very high cost for some programs is a severe impediment to its widespread use in high-performance systems.

Contributions. This paper aims to get the benefits of both pessimistic and optimistic tracking by combining them. We argue that combining pessimistic and optimistic tracking *naïvely* is insufficient for achieving sound and efficient runtime support, due to a fundamental mismatch between them. Our novel approach, called *hybrid tracking*, addresses these challenges based on insights about the interplay between dependence tracking and program synchronization. Hybrid tracking consists of two components:

1. A *hybrid state model* supports shared variables being in—and transferring between—pessimistic and optimistic *states* (i.e., handled by pessimistic and optimistic tracking, respectively).
2. An *adaptive policy* makes profile-guided decisions about when to apply pessimistic versus optimistic tracking.

We extend two kinds of runtime support to use hybrid tracking:

1. a dependence recorder, demonstrating sound *detection* of cross-thread dependences; and
2. enforcement of region serializability, demonstrating sound *controlling* of cross-thread dependences.

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

PPoPP '16 March 12–16, 2016, Barcelona, Spain

DOI: http://dx.doi.org/10.1145/2851141.2851143

We have implemented the above components and runtime support in a high-performance Java virtual machine. Our evaluation shows that although hybrid tracking's *average* performance improvement over optimistic tracking is modest, hybrid tracking (1) consistently outperforms pessimistic tracking, (2) significantly outperforms optimistic tracking for high-conflict programs, and (3) performs about the same as optimistic tracking for low-conflict programs. While pessimistic and optimistic tracking each have limitations for low- and high-conflict programs, respectively, hybrid tracking overcomes the limitations of both—suggesting it is a promising direction for efficient, flexible, software-only runtime support that targets diverse parallel software systems.

2. Background and Motivation

Runtime support that checks or enforces concurrency correctness properties must *track* cross-thread dependences, which are data dependences (write–read, write–write, and read–write dependences) involving two threads. Tracking dependences means doing one of the following *soundly* (i.e., without missing dependences):

- **Detect (monitor) dependences.** Examples: data race detectors, atomicity violation detectors, and dependence recorders (e.g., for record & replay).
- **Control (enforce) dependences.** Examples: transactional memory, enforcing memory models, and deterministic execution.

For data-race-free (DRF) executions, runtime support can track cross-thread dependences soundly by instrumenting only program synchronization operations, because shared-memory languages such as Java and C++ guarantee atomicity of synchronization-free regions for DRF executions [2, 3, 9, 27]. However, programs routinely have data races, which are hard to detect or eliminate (e.g., [12, 18, 25, 40]), so runtime support must instrument all potentially racy memory accesses. (Although sound static analysis can identify some accesses as definitely DRF, instrumenting the remaining potentially racy accesses is still expensive [15, 17, 25, 40].)

Tracking cross-thread dependences. To track cross-thread dependences, instrumentation at each memory access maintains the *last-access state* of the accessed object.[1] Without loss of generality, we assume dependence tracking uses the following per-object states:

- WrEx_T: Write exclusive for thread T. Last read or written by T.
- RdEx_T: Read exclusive for T. Last read (not written) by T.
- RdSh_c: Read shared. Last read by multiple threads. The value c helps ensure sound tracking of write–read dependences.[2]

Table 1 shows all possible state transitions, each of which is triggered by a memory access by some thread. Prior work shows that these state transitions establish happens-before edges [23] that transitively imply all of an execution's cross-thread dependences [11].

Same-state transitions involve no state change; they do not imply any cross-thread dependences. Other transitions imply potential cross-thread dependences. *Upgrading* transitions change the state to a new state that permits accesses allowed under the old state. *Fence* transitions enable detecting write–read dependences when a thread reads a RdSh_c object for the first time (prior work provides details [11], which are not integral to understanding this paper). Finally, *conflicting* transitions change the old state to a new state that disallows accesses allowed under the old state.

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

[2] Prior work that introduces the counter provides details on how it helps enable sound tracking of cross-thread dependences [11].

Transition type	Old state	Access	New state	Sync. required: Pessimistic tracking	Sync. required: Optimistic tracking
Same state	WrEx_T	R/W by T	Same	CAS	None
	RdEx_T	R by T	Same		
	RdSh_c	R by T	Same*		
Upgrading	RdEx_T	W by T	WrEx_T	CAS	CAS
	$\mathsf{RdEx}_\mathsf{T1}$	R by T2	RdSh_c*		
Fence	RdSh_c	R by T	Same*	CAS	Mem. fence
Conflicting	$\mathsf{WrEx}_\mathsf{T1}$	W by T2	$\mathsf{WrEx}_\mathsf{T2}$	CAS	Coordination
	$\mathsf{WrEx}_\mathsf{T1}$	R by T2	$\mathsf{RdEx}_\mathsf{T2}$		
	$\mathsf{RdEx}_\mathsf{T1}$	W by T2	$\mathsf{WrEx}_\mathsf{T2}$		
	RdSh_c	W by T	WrEx_T		

Table 1. All possible state transitions for last-access states. *An upgrading transition to RdSh_c gets the counter value c from a monotonically increasing global counter. A read by T of an object in the RdSh_c state requires a fence transition if and only if a per-thread counter $\mathsf{T.rdShCount} < \mathsf{c}$ [11].

Instrumentation atomicity. To track dependences accurately, instrumentation at each memory access must check, and potentially update, the accessed object's state. These actions must appear to happen together *atomically* to avoid missing dependences; we call this property *instrumentation atomicity*. Furthermore, most runtime support requires *instrumentation–access atomicity*: that the instrumentation and access appear to execute together atomically. (A notable exception is data race detection, which requires only instrumentation atomicity because it does not need to know the order of racy accesses.) In any case, instrumentation atomicity and instrumentation–access atomicity incur similar costs.

To guarantee instrumentation–access atomicity, most existing runtime support uses instrumentation that performs atomic operations at every memory access, which we call *pessimistic tracking* (Section 2.1). Alternatively, *optimistic tracking* eschews atomic operations at non-communicating accesses, but requires inter-thread coordination at some communicating accesses (Section 2.2).

We emphasize that the instrumentation and per-object states used by dependence tracking, as well as the synchronization needed to ensure instrumentation–access atomicity, are visible to runtime support only, not to programmers.

2.1 Pessimistic Tracking

Pessimistic tracking provides instrumentation–access atomicity via a small critical section around each access and its instrumentation. As Table 1 indicates, pessimistic tracking requires an atomic operation (e.g., compare-and-swap instruction) at every access. The following pseudocode shows typical instrumentation at a program *store*. (Instrumentation at a *load* is similar but more complex since there are more possible state transitions.)

```
do {
  s = o.state; // load per-object metadata
} while (s == LOCKED || !CAS(&o.state, s, LOCKED));
if (s != WrEx_T) { // T is the executing thread
  /* handle potential cross-thread dependence(s) */
}
o.f = ... ;  // program store
memfence; // type of fence depends on program access type
o.state = WrEx_T; // unlock and update metadata
```

The instrumentation starts a critical section by "locking" the object's state (represented as o.state) using a special LOCKED value.[3] If the current state is any state other than WrEx_T (T is the current executing thread), a potential cross-thread dependence exists, requiring additional runtime-support-specific work

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

(not shown). For example, a dependence recorder could record the dependence in a log, and speculation-based enforcement of region serializability could roll back and restart a code region.

Performance. Pessimistic tracking requires frequent atomic operations and memory fences, which slow program execution substantially by triggering remote cache misses and serializing out-of-order execution. In our experiments on benchmarked versions of large, real-world Java programs, pessimistic tracking (without any runtime support on top of it) slows programs by more than 4X on average (Section 7.5).

Existing runtime support commonly employs pessimistic tracking (e.g., [18–20, 24, 25]). We note that existing approaches often avoid performing an atomic operation for every memory access. For example, software transactional memory (STM) [20] can use instrumentation that avoids atomic operations for accesses to the same object in the same transaction. Some STM systems can further avoid atomic operations at loads by validating them lazily, but still require memory fences (e.g., [34]). Data race detectors [18] can avoid atomic operations for repeated accesses in the same synchronization-free region. Nonetheless, atomic operations and memory fences remain frequent enough to incur high overhead. Other approaches have sidestepped explicit dependence tracking but incur other limitations and costs, e.g., DoublePlay detects conflicts implicitly using speculation and replication, but it adds high overhead unless extra cores are available [38].

2.2 Optimistic Tracking

In contrast, optimistic tracking avoids synchronization at most accesses. Prior work uses optimistic tracking either to implement program locks (Section 8) [13, 22, 33] or to track cross-thread dependences [11, 35, 39]. This paper focuses on the latter context.

Optimistic tracking provides instrumentation–access atomicity without requiring synchronization at accesses that trigger no state change, but it requires coordination at accesses that trigger conflicting state changes. Table 1 shows the differing kinds of synchronization needed for each transition type. The following pseudocode shows the instrumentation added at a program *store* (instrumentation for a *load* is similar but more complex):

```
if (o.state != WrEx_T) { // fast path
  slowPath(o);
}
o.f = ... ; // program store
```

If the object's state is already WrEx_T, the instrumentation takes the synchronization-free *fast path*. Otherwise, the instrumentation executes the *slow path*, shown in Figure 1, which changes the state and handles the possible cross-thread dependence. Upgrading transitions require an atomic operation to avoid racing with other threads changing the state. Fence transitions require a memory fence to ensure visibility for write–read dependences.

Conflicting transitions require coordination. Conflicting transitions (last four rows of Table 1) require that threads *coordinate* with each other, in order to ensure that thread(s) do not continue performing *unsynchronized* same-state transitions to the object. Figure 1 shows the instrumentation slow path, for a program store only. To initiate coordination, the executing thread T first changes the object's state to an *intermediate* state Int_T (line 8), which simplifies the protocol by allowing only one thread at a time to initiate coordination for an object. T then coordinates with the remote thread (line 12) to ensure that T's state change does not interrupt the remote thread's instrumentation–access atomicity.[4]

The remote thread, which we call remoteT, participates in coordination only when it is at a *safe point*: a program point that does

[4] If o.state is RdSh_c, T conservatively coordinates with every other thread.

```
1  slowPath(o) {
2    state = o.state;
3    if (state == RdEx_T) {
4      ... ; // upgrading transition to WrEx_T
5      return;
6    }
7    // Coordination for conflicting transition:
8    while (state == Int_* || !CAS(&o.state, state, Int_T)) {
9      checkAndRespondToRequests(); // non-blocking safe point
10     state = o.state; // re-read state
11   }
12   coordinate(getOwner(state));
13   o.state = WrEx_T;
14 }

15 coordinate(remoteT) {
16   response = sendRequest(remoteT); // return true if implicit
         coordination used
17   while (!response) {
18     checkAndRespondToRequests(); // non-blocking safe point
19     response = checkResponse(remoteT);
20   }
21 }
```

Figure 1. Pseudocode for optimistic tracking's instrumentation slow path (for program stores only) and coordination. T is the executing thread.

not interrupt instrumentation–access atomicity (not shown in the figure). Conveniently, managed language VMs already place safe points at periodic points in compiled code (e.g., method entries and loop back edges) so threads can be stopped promptly, e.g., for stop-the-world garbage collection. Blocking operations, such as waiting to acquire a lock or for I/O, are also safe points. If remoteT is at a blocking safe point, T coordinates with remoteT *implicitly* by updating remoteT's status atomically, which remoteT will see when it finishes blocking. Otherwise, T coordinates with remoteT *explicitly*: T sends a request to remoteT, and remoteT responds at its next safe point. (Figure 1 does *not* show the actions of remoteT.) Whenever a safe point responds (implicitly or explicitly) to coordination request(s), it is called a *responding safe point*. An important detail is that while T waits for an explicit coordination response, it acts as a safe point (line 18), so other threads can perform coordination with T in order to gain access to other objects, thus avoiding deadlock.

Finally, T changes the state to WrEx_T (line 13) and proceeds with its access. Since remoteT coordinates only at a safe point, and T does not proceed with its access until coordination completes, instrumentation–access atomicity is preserved.

Performance. Optimistic tracking exploits a tradeoff: it avoids synchronization in the common, non-conflicting case but requires coordination in the uncommon, conflicting case. As Section 7.5 shows, for programs that perform little communication, optimistic tracking incurs low overhead. For programs that perform more communication (e.g., as little as 0.5% of accesses conflicting), optimistic tracking incurs high overhead (e.g., >100% run-time overhead). Optimistic tracking's key limitation—and the main impediment to its widespread use—is its poor performance for all but low-conflict executions.

The following table reports costs of different kinds of state transitions, averaged across all programs (Section 7.2 describes overall experimental methodology):

	Pessimistic	**Optimistic**		
		Same state	Conflicting	
			Explicit	Implicit
CPU cycles	150	47	9,200	360

The average time in CPU cycles for pessimistic instrumentation is 150 cycles, which is largely independent of the transition type. Optimistic instrumentation's cost is only a few dozen cycles for non-communicating accesses (*Same state*), but conflicting transitions that use *Explicit* coordination cost 2–3 orders of magnitude more by incurring the latency of roundtrip communication. *Implicit* coordination requires atomic operations but incurs no latency, so its cost is relatively close to the cost of a pessimistic access.

Goal and outline. Our goal is to develop a hybrid of pessimistic and optimistic tracking that keeps overhead low by using optimistic tracking for most accesses, but avoids most coordination by using pessimistic tracking for most conflicting accesses. Sections 3 presents challenges inherent in combining pessimistic and optimistic tracking, and introduces a hybrid state model that addresses these challenges. Sections 4 and 5 design sound and efficient runtime support using the hybrid state model. Section 6 describes a policy that decides between pessimistic and optimistic states at run time. The remaining sections describe our implementation and evaluation, and compare with related work.

3. Hybrid State Model

This section introduces a hybrid state model that combines the state models of pessimistic and optimistic tracking. Section 3.1 argues that hybridization presents fundamental challenges, and then describes insights for addressing these challenges. Section 3.2 presents details of the hybrid state model.

3.1 The Pessimistic–Optimistic Mismatch

Pessimistic and optimistic tracking are fundamentally different in two key ways that complicate hybridization. First, pessimistic and optimistic tracking differ in how they transfer access privileges. Pessimistic tracking unlocks an object's state after a program access, allowing another thread to lock the state. Optimistic tracking, on the other hand, does *not* unlock the state after an access; instead, a thread relinquishes access privileges only when requested by another thread. To support objects being in both pessimistic and optimistic states, it seems that each access must be followed by potentially costly instrumentation that *conditionally* unlocks the state (depending on whether the state is pessimistic).

Second, pessimistic and optimistic tracking provide instrumentation–access atomicity differently. Pessimistic tracking provides atomicity of each instrumentation–access pair. Optimistic tracking provides atomicity interrupted at responding safe points—including conflicting accesses that respond to coordination requests. This mismatch implies that the atomicity of instrumented code can be interrupted at points that are statically unpredictable, making it problematic to design efficient runtime support that detects and controls cross-thread dependences. This problem is easier to understand in the context of specific kinds of runtime support; Sections 4 and 5 explain these challenges in the contexts of the dependence recorder and region serializability (RS) enforcer.

In the early stages of this work, we designed and implemented a straightforward approach for combining pessimistic and optimistic tracking. This approach added conditional instrumentation after every program access, to unlock the state when it was pessimistic. We built a dependence recorder and RS enforcer on top of this hybrid approach, but they added significant overhead to perform conditional instrumentation and to deal with atomicity being interrupted unpredictably at many program points.

To overcome the mismatch between pessimistic and optimistic tracking that impaired our initial design, we introduce the following insight: the hybrid state model can and should *defer unlocking* of pessimistic states. Deferring unlocking consists of the following design points:

```
            T1                                    T2
synchronized (m) {
  ...
  /* lock o.state */
  o.f = ... ;
  ...
  /* unlock all states */
}
                                     synchronized (m) {
                                       ...
                                       /* lock o.state */
                                       ... = o.f;
                                       ...
```

(a) For well-synchronized accesses, locking a pessimistic state encounters no contention.

```
            T1                                    T2
synchronized (m) {
  ...
  /* lock o.state */
  o.f = ... ;             <-------------   /* lock o.state */
  ...                                        .
  ...                                        .
  /* safe point */        ------------->     .
  ...                                      ... = o.f;
```

(b) An access involved in an (object-level) data race may encounter contention. In this case, hybrid tracking triggers coordination.

Figure 2. Deferred unlocking encounters contention only for object-level data races. Comments show instrumentation actions assuming o is in pessimistic states.

- A thread defers unlocking pessimistic states until the next *program synchronization release operations* (PSRO) such as lock release, monitor wait, or thread fork.
- To avoid substantial false contention from concurrent readers, pessimistic states use *reader–writer locking*.
- A thread encountering any remaining contention "falls back" to using *coordination* to change an object's state.

Interestingly, if instrumentation encounters contention trying to lock a pessimistic state, the access must be involved in an *object-level data race*: two unsynchronized, conflicting accesses to the same object, but not necessarily the same field or array element. An object-level data race is a necessary but insufficient condition for a true (precise) data race. Prior work shows that object-level data races closely over-approximate precise data races in practice [39]. The performance of our hybrid design relies on object-level data races being rare, so that few (if any) pessimistic transitions encounter contention.

Deferring unlocking bridges the pessimistic–optimistic mismatch by making pessimistic tracking more "optimistic": threads do not unlock pessimistic states until PSROs, but incur high coordination cost (the same as for optimistic states) if a conflicting access occurs in the meantime.

Example. Figure 2 illustrates deferring unlocking of pessimistic states. The example assumes o is in pessimistic states for the accesses shown. In Figure 2(a), each thread executes a critical section acquiring the same program lock m. Code comments (e.g., */* lock o.state */*) summarize the run-time behavior of hybrid tracking's instrumentation. Immediately before T1 releases m (a PSRO), instrumentation unlocks all pessimistic states that T1 has locked, including o's state. T2 thus locks o's state without contention.

In contrast, in Figure 2(b), the two accesses are involved in an object-level data race (in this case, a true data race). As a result, T2 encounters contention when trying to lock o's state. T2 handles

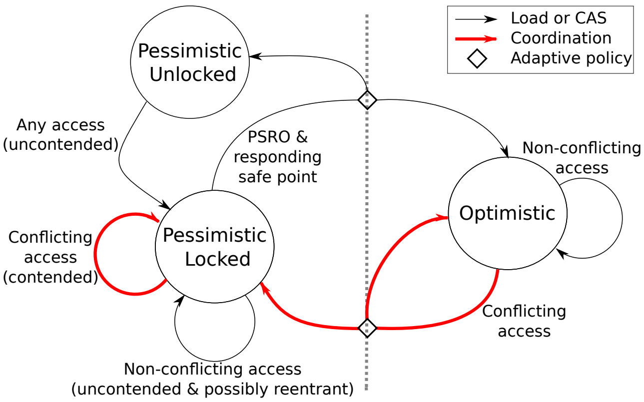


Figure 3. High-level state transition diagram for the hybrid state model. The left and right halves show transitions starting in pessimistic and optimistic states, respectively. The diamonds on the vertical dashed line indicate decisions by the adaptive policy, described in Section 6.

this case safely by falling back to using coordination: T2 sends a coordination request to T1, which unlocks all pessimistic states at the next responding safe point, enabling T2 to lock o's state.

3.2 States, Terminology, and Transitions

The hybrid state model uses the following states:

- Pessimistic states can be either unlocked or locked. The *pessimistic unlocked* states are $WrEx_T^{Pess}$, $RdEx_T^{Pess}$, and $RdSh_c^{Pess}$. The *pessimistic locked* states are $WrEx_T^{RLock}$, $WrEx_T^{WLock}$, $RdEx_T^{RLock}$, and $RdSh_c^{RLock(n)}$. To support reader–writer locking, a $WrEx_T$ state can be either read- or write-locked, and a $RdSh_c^{RLock(n)}$ state is read-locked by n threads. The read-locked write-exclusive state ($WrEx_T^{RLock}$) enables a second concurrent reader to upgrade to $RdSh_c^{RLock(2)}$, instead of encountering contention. To support reentrant read locks, each thread also keeps track of the set of objects whose states it has read-locked.
- The optimistic states are $WrEx_T^{Opt}$, $RdEx_T^{Opt}$, and $RdSh_c^{Opt}$.

A *pessimistic (or optimistic) object* is an object whose state is pessimistic (optimistic). A *pessimistic (optimistic) access* is a program access to a pessimistic (optimistic) object. A *pessimistic (optimistic) transition* is a transition from a pessimistic (optimistic) state to another pessimistic (optimistic) state. The model also supports transitions *between* pessimistic and optimistic states.

Figure 3 shows at a high level the state transitions in the hybrid state model. The labeled circles summarize the three types of states: pessimistic unlocked, pessimistic locked, and optimistic. Arrows represent transitions between states: bold, red arrows show transitions requiring coordination; other transitions do not require coordination. The rest of this section further explains Figure 3, focusing on transitions that are different from those shown in Table 1. Appendix A shows pseudocode for hybrid tracking's instrumentation. Appendix B presents a table detailing *every* state transition.

Pessimistic uncontended transitions. Any access to an unlocked pessimistic state triggers an *uncontended* transition to a corresponding locked state (see the transition labeled "Any access (uncontended)" in Figure 3). For example, a read (or write) by T1 to an object in $WrEx_{T1}^{Pess}$ state triggers an uncontended transition to $WrEx_{T1}^{RLock}$ ($WrEx_{T1}^{WLock}$). A read by T2 to an object in $WrEx_{T1}^{Pess}$ triggers an uncontended transition to $RdEx_{T2}^{RLock}$.

An access to a locked state that *does not conflict* with the state also triggers an uncontended transition (transition labeled "Non-conflicting access (uncontended & possibly reentrant)"). For example, a read by T2 to a $RdEx_{T1}^{RLock}$ object triggers an uncontended transition to $RdSh_c^{RLock(2)}$ (read-locked by T1 and T2). A write by T1 to a $WrEx_{T1}^{RLock}$ object triggers an uncontended transition to $WrEx_{T1}^{WLock}$. If an uncontended transition requires no state change at all (e.g., a read by T1 to an object in $RdEx_{T1}^{RLock}$ state), we also call the transition *reentrant*. Reentrant transitions require no atomic operations.

Unlocking of pessimistic states. To support deferred unlocking, each thread records every pessimistic object whose state it has locked in the thread's *lock buffer*. Every program synchronization release operation (PSRO) and responding safe point *flushes* the buffer by unlocking the states of all objects in the buffer (transition labeled "PSRO & responding safe point"). Unlocking a $RdSh_c^{RLock(n)}$ object means transitioning to $RdSh_c^{RLock(n-1)}$ (if $n>1$) or the unlocked state $RdSh_c^{Pess}$ (if $n=1$). Whenever a thread flushes its lock buffer, it also clears its set of read-locked objects.

Pessimistic contended transitions. An access that *conflicts* with a pessimistic locked state cannot immediately change the state. It triggers a *contended* state transition, which initiates coordination with the thread(s) that have locked the object's state (transition labeled "Conflicting access (contended)").

Since every responding safe point flushes the lock buffer, the thread(s) that have locked the state will unlock it, allowing the accessing thread to change the state into a compatible pessimistic locked state. By using coordination to trigger early unlocking of states, contended transitions ensure responsiveness and deadlock freedom when an execution violates deferred unlocking's assumption of object-level data race freedom.

As an example, in Figure 2(b), a read by T2 to an object in $WrEx_{T1}^{WLock}$ triggers a contended transition: T1 unlocks the state to $WrEx_{T1}^{Pess}$ before responding to coordination. T2 then performs an uncontended transition from $WrEx_{T1}^{Pess}$ to $RdEx_{T2}^{RLock}$.

Transitions between pessimistic and optimistic states. The model supports transitioning to an optimistic state whenever it unlocks a pessimistic state (upper diamond in Figure 3), and to a pessimistic state from an optimistic state on any conflicting transition (lower diamond).

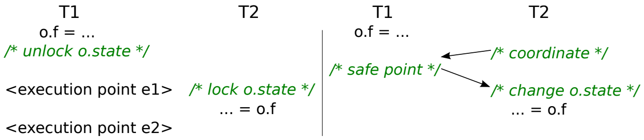
Although we have designed and presented hybrid tracking based on the states and transitions in Table 1, our hybridization approach could in theory be applied to other optimistic and pessimistic approaches that use different state models to track dependences.

4. Recording and Replaying Dependences

This section demonstrates how runtime support that needs to *detect* (i.e., monitor) cross-thread dependences soundly can use our hybrid state model. We build a *dependence recorder* based on hybrid tracking that identifies and records happens-before edges that transitively imply all cross-thread dependences in the execution.

4.1 Optimistic Dependence Recorder and Replayer

Multithreaded record & replay helps programmers debug nondeterministic multithreaded programs, and it provides systems benefits such as replication-based fault tolerance [24–26, 30, 32, 38, 41]. Prior work introduces a record & replay approach that designs (1) an *optimistic recorder* on top of optimistic tracking and (2) an *optimistic replayer* for the recorder [10, 11]. (The optimistic replayer is "optimistic" because it replays dependences recorded by the optimistic recorder. It does *not* use optimistic tracking.) The optimistic recorder identifies and records happens-before edges at transitions between $WrEx^{Opt}$, $RdEx^{Opt}$, and $RdSh^{Opt}$ states. It records each happens-before edge by recording its *source* and *sink* in per-thread logs. In another execution, the optimistic replayer replays each



(a) pessimistic transition (b) optimistic transition

Figure 4. The challenge of recording pessimistic conflicting transitions.

happens-before edge by making the sink wait for its corresponding source to be reached.

4.2 Hybrid Dependence Recorder & Replayer

We design a *hybrid recorder* based on hybrid tracking, and a *hybrid replayer* for the hybrid recorder. For optimistic transitions, the hybrid recorder uses the same approach as the optimistic recorder. For some, but not all, pessimistic transitions, the hybrid recorder uses essentially the same approach as for optimistic transitions, since pessimistic and optimistic states and transitions each maintain the same last-access information. For example, the recorder can record a happens-before edge for $\mathsf{RdEx}_{\mathsf{T}}^{\mathsf{Pess}} \rightarrow \mathsf{RdSh}_{\mathsf{c}}^{\mathsf{RLock(2)}}$ in the same way that it records $\mathsf{RdEx}_{\mathsf{T}}^{\mathsf{Opt}} \rightarrow \mathsf{RdSh}_{\mathsf{c}}^{\mathsf{Opt}}$.

Pessimistic conflicting transitions. The key challenge is pessimistic transitions that involve conflicting states, as Figure 4(a) shows. In this example, suppose pessimistic transitions *do not* defer unlocking. Thread T1 immediately unlocks an object o to $\mathsf{WrEx}_{\mathsf{T1}}^{\mathsf{Pess}}$ state after a write to o; then T2 wants to read o. It is challenging to identify and record the *source* of the happens-before edge, because T1 continues executing during the pessimistic transition by T2. An eligible source needs to be (1) *after* T1's write to o, in order to capture the cross-thread dependence soundly, but (2) *no later than* T1's current execution point e1, or else replay could deadlock: suppose T2 records a future execution point e2, and T1 writes to o again (not shown) between e1 and e2. T1 would record an execution point *after* T2's read of o as the source of another happens-before edge, creating a cycle of dependences.

In contrast, an *optimistic* conflicting transition triggers coordination, as shown in Figure 4(b). T1 stops to respond to T2 at a safe point, providing an opportunity to record the happens-before source. The responding safe point satisfies both requirements for an eligible source.

The hybrid recorder could record every pessimistic access, but they are frequent enough that recording each one would be expensive. Alternatively, incrementing a counter at every pessimistic access would be efficient—but the replayed run would not know which accesses had been pessimistic versus optimistic during the recorded run. We encountered these challenges in our initial design of the hybrid recorder (Section 3.1), which performed worse on average than the optimistic recorder.

Utilizing deferred unlocking. These challenges are naturally addressed by, and thus motivate the use of, deferred unlocking (Section 3.1). By deferring unlocking of pessimistic states until program synchronization release operations (PSROs), the potential sources of happens-before edges are effectively limited.

The hybrid recorder handles pessimistic *uncontended* transitions involving conflicting states as follows. In both recorded and replayed executions, instrumentation at every PSRO and responding safe point increments a per-thread *release counter*. Using Figure 2(a) from Section 3.1 as an example, T1 increments its release counter *before* it releases the program lock m. When T2 changes the state to $\mathsf{RdEx}_{\mathsf{T2}}^{\mathsf{RLock}}$, it records the happens-before edge in its log by reading T1's release counter and recording its value. Since each PSRO and responding safe point has release semantics, and each state change has acquire semantics, T2 is guaranteed to read

```
<region boundary>
/* possibly lock o.state */
... = o.f;
...
/* possibly lock p.state */
p.g = ... ;
...
/* possibly unlock o.state , p.state , ... */
<region boundary>
```

Figure 5. Challenge of building an RS enforcer using hybrid tracking.

a value of T1's release counter that is at least as great as the value at the first PSRO *after* T1 writes to o. In addition, T2 cannot read a value that T1's release counter has not reached, preventing deadlock during replay. During replay, T2 waits for T1's release counter to reach the recorded value.

For a *contended* transition as in Figure 2(b), T2 initiates coordination. T1 unlocks o's state to $\mathsf{WrEx}_{\mathsf{T1}}^{\mathsf{Pess}}$, responds at a safe point, and records the response just as it would record an *optimistic* coordination response. T2 then records its uncontended transition from $\mathsf{WrEx}_{\mathsf{T1}}^{\mathsf{Pess}}$ to $\mathsf{RdEx}_{\mathsf{T2}}^{\mathsf{RLock}}$ as described above.

5. Enforcing Region Serializability

This section applies the hybrid state model to enforcing serializability (atomicity) of executed code regions, demonstrating how the model enables *controlling* cross-thread dependences.

5.1 Optimistic RS Enforcer

Modern language memory models make strong guarantees for data-race-free (DRF) executions but provide virtually no guarantees for racy executions [2, 3, 8, 9, 27]. Prior work enforces memory models that provide *region serializability* (RS) even for executions with data races [29, 36]. We focus on work that introduces a memory model called *statically bounded region serializability* (SBRS) that provides serializability of regions that are bounded by program synchronization operations, method calls, and loop back edges [36].

Prior work, which we call the *optimistic enforcer*, enforces SBRS using optimistic tracking at each object access [36]. The optimistic enforcer provides region serializability via two-phase locking: each object access uses optimistic tracking to change the state if needed, and a region does not relinquish objects' states (i.e., does not respond to coordination requests) until the region ends. However, to avoid deadlock, a thread may respond to coordination requests while itself waiting to complete a transition (lines 9 and 18 in Figure 1 from Section 2.2), relinquishing ownership of objects' states and thus potentially violating serializability.

The optimistic enforcer transforms regions at compile time so they can restart safely after responding to a coordination request.

5.2 Hybrid RS Enforcer

To understand the challenges of using hybrid tracking for the RS enforcer, consider how an RS enforcer based on *pessimistic tracking* would work. To preserve serializability, no pessimistic state locked during a region's execution should be unlocked until the region completes. At region end, instrumentation should unlock each pessimistic state locked during the region's execution.

However, using *hybrid* tracking presents a challenge, as illustrated in Figure 5. The compiler cannot predict whether the accesses to objects o and p will use pessimistic versus optimistic tracking, so each region end needs conditional checks for which pessimistic states to unlock, if any. Since we expect most accesses to be optimistic, most regions would need to unlock *no* pessimistic states. As statically bounded regions are short, the overhead of checking at the end of each region would be significant. We encountered these challenges in our initial design of a hybrid enforcer (Section 3.1).

Using deferred unlocking. Our *hybrid enforcer* relies on deferred unlocking to address these challenges. Hybrid tracking defers unlocking of pessimistic states until program synchronization release operations (PSROs). PSROs are generally infrequent compared to region boundaries, so it is inexpensive to flush the lock buffer at each PSRO. Regions thus unlock pessimistic states only at region boundaries, preserving SBRS.

The one exception is pessimistic *contended* transitions, which trigger coordination in the middle of a region. Since the thread initiating coordination can respond to other threads' coordination requests, a region restarts after completing coordination, just as it does for *optimistic* conflicting transitions.

6. Adaptive Policy

This section addresses how to choose between pessimistic and optimistic states at run time. We introduce a *cost–benefit model* for deciding whether an object should be in pessimistic or optimistic states, and an efficient *policy* that approximates the cost–benefit model based on online profiling.

6.1 Cost–Benefit Model

The basic idea of the cost–benefit model is that an object's state should be pessimistic (versus optimistic) if and only if the total time incurred on optimistic transitions for the object would exceed the total time incurred on pessimistic transitions.

A limitation of our cost–benefit model is that it models pessimistic transitions based on pessimistic tracking *without deferred unlocking*. Thus, the model assumes that all accesses to objects in optimistic states that trigger conflicting transitions (and thus coordination), would trigger *uncontended* (and thus coordination-free), *non-reentrant* pessimistic transitions if the objects were in pessimistic states.

The cost–benefit model considers each object individually. Let N_{pess} be the number of pessimistic transitions that *would* occur for the object if its state were always pessimistic. N_{pess} thus counts all program accesses to an object. Let N_{confl} and $N_{nonConfl}$ be the numbers of conflicting and non-conflicting transitions, respectively, that would occur if the state were optimistic. Since together N_{confl} and $N_{nonConfl}$ count all accesses,

$$N_{pess} = N_{nonConfl} + N_{confl} \tag{1}$$

Let $T_{nonConfl}$, T_{confl}, and T_{pess} be the average time costs for an optimistic non-conflicting,[5] optimistic conflicting,[6] and pessimistic transition, respectively. The model considers these values to be (platform-specific) constants computed ahead of time, e.g., from the table in Section 2.2. An object's state should be optimistic if and only if the following is true:

$$T_{pess} \times N_{pess} \geq T_{nonConfl} \times N_{nonConfl} + T_{confl} \times N_{confl} \tag{2}$$

The left-hand side of (2) is the total time spent on state transitions if the object's state were pessimistic. The right-hand side is the total time on state transitions if the state were optimistic.

Applying (1) into (2) and transforming it yields:

$$N_{nonConfl} \geq K_{confl} \times N_{confl} \tag{3}$$

[5] The model computes the time for non-conflicting transitions as simply the time for same-state transitions, ignoring other non-conflicting transitions (upgrading and fence transitions), which each incur a cost similar to a pessimistic transition's cost.

[6] T_{confl} is the time for a conflicting transition using *explicit* coordination.

where K_{confl} is a run-time constant:

$$K_{confl} = \frac{T_{confl} - T_{pess}}{T_{pess} - T_{nonConfl}}$$

Thus, according to (3), using the cost–benefit model requires knowing only the numbers of non-conflicting and conflicting transitions ($N_{nonConfl}$ and N_{confl}), or merely their ratio.

6.2 Profile-Guided Adaptive Policy

Using the cost–benefit model to change each object's state to optimistic or pessimistic at run time presents several challenges that we address as follows.

Predicting the future. The cost–benefit model seems to require oracle knowledge: it needs to know the future ratio $N_{nonConfl}/N_{confl}$ when allocating an object, to initialize its state. The adaptive policy instead uses *online profiling*, assuming future behavior approximates past behavior in the same execution. Each object newly allocated by thread T starts in the $\mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Opt}}$ state.

Profiling each object separately might limit the adaptive policy's effectiveness. For example, if many objects each trigger only a few conflicting transitions, the policy will not transfer them to pessimistic states early enough. Profiling objects in *aggregate* (e.g., by object type) could enable allocating certain objects directly into pessimistic states. However, for our evaluated workloads, our policy gets nearly all of the possible benefit (Section 7.3).

Efficient profiling. Counting optimistic same-state transitions would be expensive because they are common (by design). The profiling thus counts only conflicting transitions for optimistic objects,[7] but it counts all pessimistic transitions, since they are relatively infrequent (by design). This policy thus readily transfers potentially high-conflict objects to pessimistic states—at which point more-intrusive profiling categorizes every pessimistic transition in order to determine whether an object should stay in pessimistic states or change back to optimistic states.

For each object o, the profiling counts the number of optimistic conflicting transitions $\mathsf{o.numConflicts}$. If an object experiences "enough" conflicting transitions, i.e., if

$$\mathsf{o.numConflicts} \geq \mathit{Cutoff}_{confl} \tag{4}$$

then the policy transitions the object to a pessimistic state.

For every pessimistic transition, profiling counts whether it was non-conflicting or conflicting. The policy changes an object back to optimistic based on the following formula, derived from (3):

$$N_{nonConfl} \geq K_{confl} \times N_{confl} + \mathit{Inertia} \tag{5}$$

The parameter $\mathit{Inertia}$ avoids prematurely changing back to optimistic states before a significant amount of profiling has occurred.

Checks and balances. By using a low value for Cutoff_{confl}, the adaptive policy quickly transitions objects to pessimistic states if they *might* be better off in pessimistic states, based on (4). Then profile-guided decisions based on (5) can more accurately distinguish objects that should be in pessimistic versus optimistic states. To avoid repeatedly switching an object between optimistic and pessimistic states that should ideally remain optimistic, the policy disallows repeated transitions to pessimistic: each object starts in $\mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Opt}}$ state; it can transition to pessimistic and later can transition back to optimistic; after that, it must stay optimistic. Alternatively, the policy could allow repeated transitions from optimistic to pessimistic, but with a greater Cutoff_{confl} value.

[7] The policy counts only transitions that use explicit coordination, since implicit coordination is roughly as expensive as a pessimistic transition.

	Optimistic transitions				Pessimistic transitions			Opt. to	Pess.
	Same state		Conflicting		Uncontended	%Reentrant	Contended	Pess.	to Opt.
eclipse6	(1.2×10^{10})	1.2×10^{10}	(1.3×10^{5})	1.3×10^{5}	1.5×10^{6}	32%	1.3×10^{2}	1.2×10^{2}	1.1×10^{2}
hsqldb6	(6.1×10^{8})	6.1×10^{8}	(9.2×10^{5})	5.2×10^{5}	4.7×10^{6}	64%	9.0×10^{2}	5.1×10^{1}	0–1
lusearch6	(2.4×10^{9})	2.3×10^{9}	(4.4×10^{3})	4.3×10^{3}	2.6×10^{2}	30%	0	1.0×10^{0}	0
xalan6	(1.1×10^{10})	1.0×10^{10}	(1.8×10^{7})	3.9×10^{5}	2.1×10^{8}	52%	1.5×10^{1}	5.4×10^{2}	1.0×10^{2}
avrora9	(6.0×10^{9})	6.0×10^{9}	(6.0×10^{6})	2.7×10^{6}	8.4×10^{6}	17%	8.0×10^{5}	1.0×10^{5}	1.2×10^{2}
jython9	(5.1×10^{9})	5.1×10^{9}	(6.7×10^{1})	7.3×10^{1}	0	0%	0	0	0
luindex9	(3.4×10^{8})	3.4×10^{8}	(3.7×10^{2})	3.8×10^{2}	0	0%	0	0	0
lusearch9	(2.3×10^{9})	2.3×10^{9}	(2.8×10^{3})	2.3×10^{3}	3.9×10^{3}	44%	7.6×10^{1}	1.1×10^{1}	2.0×10^{0}
pmd9	(5.6×10^{8})	5.5×10^{8}	(4.2×10^{4})	1.7×10^{4}	1.9×10^{5}	58%	2.1×10^{3}	3.0×10^{2}	5.4×10^{1}
sunflow9	(1.7×10^{10})	1.7×10^{10}	(6.1×10^{3})	6.2×10^{3}	5.9×10^{3}	92%	3.0×10^{1}	8.4×10^{0}	3.6×10^{0}
xalan9	(1.0×10^{10})	9.8×10^{9}	(1.7×10^{7})	2.9×10^{5}	1.9×10^{8}	68%	3.0×10^{1}	9.0×10^{2}	1.4×10^{2}
pjbb2000	(1.7×10^{9})	1.7×10^{9}	(9.5×10^{5})	9.3×10^{5}	2.4×10^{6}	58%	1.3×10^{2}	2.4×10^{3}	1.1×10^{3}
pjbb2005	(6.6×10^{9})	6.5×10^{9}	(4.4×10^{7})	8.4×10^{5}	1.4×10^{8}	32%	7.6×10^{5}	3.2×10^{3}	3.1×10^{3}

Table 2. State transitions for hybrid tracking, compared with state transitions for optimistic tracking alone (shown in parentheses).

7. Evaluation

This section evaluates the run-time characteristics and performance of hybrid tracking, compared with pessimistic and optimistic tracking alone. It also compares the performance of the hybrid and optimistic versions of the dependence recorder and RS enforcer.

7.1 Implementation

We have implemented the hybrid state model, adaptive policy, hybrid dependence recorder and replayer, and hybrid RS enforcer in Jikes RVM 3.1.3, a Java virtual machine [4] that performs competitively with commercial JVMs [6]. We have made our implementation, which targets the IA-32 platform, publicly available on the Jikes RVM Research Archive. Our implementation builds on publicly available implementations of pessimistic and optimistic tracking [11], the optimistic recorder and replayer [10], and the optimistic RS enforcer [36].

By targeting a managed language, our implementation can piggyback on existing language implementation features. Notably, coordination piggybacks on the safe point mechanism that commonly exists in managed language implementations. An implementation for a native language would need to add support for safe points.

Jikes RVM's dynamic just-in-time compilers insert instrumentation before every memory access, PSRO, and safe point in the application and Java libraries. The implementation adds two 32-bit words to each (scalar and array) object and static field: one for last-access state and another for the adaptive policy's profile information. For exclusive states ($\mathsf{WrEx}_\mathsf{T}^*$ and $\mathsf{RdEx}_\mathsf{T}^*$), the state word encodes T's (8-byte-aligned) address and uses remaining bits to differentiate states (e.g., pessimistic versus optimistic; WrEx versus RdEx). For $\mathsf{RdSh}_\mathsf{c}^*$ states, the bits encode c and the read-lock count, and differentiate pessimistic versus optimistic.

Extraneous contention. Due to limited bit patterns available in a metadata word, our prototype implementation omits the $\mathsf{WrEx}_\mathsf{T}^{\mathsf{RLock}}$ state: a read to a $\mathsf{WrEx}_\mathsf{T}^{\mathsf{Pess}}$ object triggers a transition to $\mathsf{WrEx}_\mathsf{T}^{\mathsf{WLock}}$. The implementation could avoid this limitation with more engineering effort, e.g., by encoding an identifier for T, rather than T's address, for $\mathsf{WrEx}_\mathsf{T}^{\mathsf{Pess}}$ and $\mathsf{RdEx}_\mathsf{T}^{\mathsf{Pess}}$ states.

Thus, the implementation may encounter pessimistic contention even in the absence of object-level data races. Suppose T1 reads an object in $\mathsf{WrEx}_{\mathsf{T1}}^{\mathsf{Pess}}$ state, transitioning the state to $\mathsf{WrEx}_{\mathsf{T1}}^{\mathsf{WLock}}$. T2 then reads the object, triggering a pessimistic contended transition. However, T1 has only *read* the object since its last PSRO, i.e., no object-level data race exists in this case.

To measure potential costs incurred by triggering unnecessary coordination, we implemented and evaluated an alternate configuration in which a read of a $\mathsf{WrEx}_{\mathsf{T1}}^{\mathsf{Pess}}$ object by T1 triggers a transition to $\mathsf{RdEx}_{\mathsf{T1}}^{\mathsf{RLock}}$. This configuration triggers coordination only when object-level data races exist, but it loses information about T1's previous write to the object, making it unsuitable for runtime support that needs to detect cross-thread dependences soundly. This *unsound* configuration provided no performance benefit, indicating that the *default* configuration is not encountering significant spurious contention in our experiments.

Optimistic tracking performance issue. When investigating the performance of high-conflict microbenchmarks (Section 7.5), we discovered an optimization opportunity that improves the performance of the optimistic tracking implementation. In particular, releasing a program lock that is a so-called "fat" lock [5] can incur significant latency, so making this operation a blocking safe point improves performance significantly. Our experiments do not include this optimization. We discovered the optimization shortly before the camera-ready deadline, and including it would require re-tuning the adaptive policy and re-collecting all results.

7.2 Methodology

Our experiments execute benchmarked versions of real applications: the DaCapo benchmarks, versions 2006-10-MR2 and 9.12-bach (2009) [7] (limited to multithreaded programs that Jikes RVM can run), and fixed-workload versions of SPECjbb2000 and 2005.[8]

The experiments run on a system with four Intel Xeon E5-4620 8-core processors (32 cores total) running Linux 2.6.32. We build a high-performance configuration (FastAdaptiveGenImmix) of Jikes RVM. Each performance result is the median of 20 trial runs; we also show the mean as the center of 95% confidence intervals. Each reported statistic is the mean from five statistics-gathering runs.

7.3 Adaptive Policy Limit Study

To evaluate whether per-object profiling identifies most optimistic conflicting transitions in advance, we perform a limit study on optimistic tracking alone. Figure 6 plots a cumulative distribution of the number of optimistic conflicting transitions (explicit coordination only) triggered by each object. For each point (x, y), y counts total conflicting transitions—as a percentage of *all* accesses—involving objects that have (so far) triggered at most x conflicting transitions. For example, (4, 0.05%) means that 0.05% of all accesses triggered conflicting transitions that were the first, second, third, or fourth conflicting transition triggered by the accessed object. The maximum y value for each program is its overall rate of conflicting transitions (explicit coordination only).

The plot shows that, at least for these programs, each object's first few conflicting transitions together constitute an insignificant

[8] `http://spec.org/`, `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`

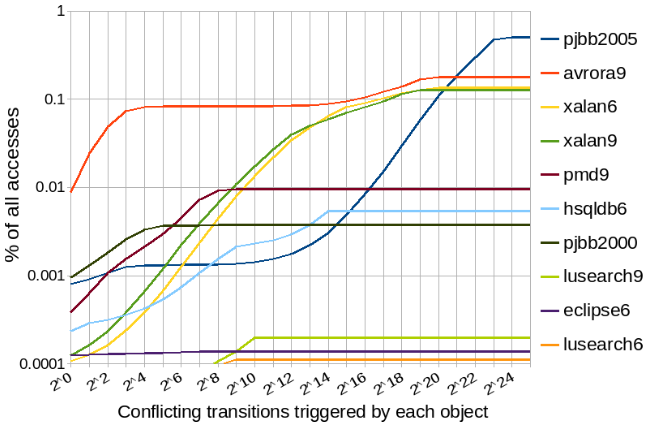


Figure 6. Cumulative distribution of conflicting transitions (explicit coordination only) triggered per object for optimistic tracking. Both axes use a logarithmic scale. The legend sorts programs by their maximum y-axis value. Three programs have a conflict rate $<$0.0001% and are excluded.

fraction of overall program accesses. For high-conflict programs, most conflicting transitions are to objects that have triggered many conflicting transitions (avrora9 is an exception). For low-conflict programs, the overall conflict rate is low, so conflicting transitions are negligible. Thus, per-object profiling can "catch" most conflicting accesses, leaving little additional opportunity for aggregate profiling.

The rest of the paper's experiments use the following adaptive policy parameter values: $Cutoff_{confl}$ = 4, K_{confl} = 200, $Inertia$ = 100. We find that larger values of $Cutoff_{confl}$ have little impact (results not shown), except for avrora9, as Figure 6 would suggest. Performance is not very sensitive to the other parameters; various values for K_{confl} (20–1,600) and $Inertia$ (20–1,600) are effective.

7.4 Run-Time Characteristics

Table 2 counts state transitions under hybrid tracking. The table breaks down *Optimistic transitions* into *Same state* and *Conflicting* transitions, which have significantly different costs (Section 2.2). For comparison, transitions triggered under optimistic tracking alone are shown in parentheses.

The *Conflicting* column measures how well the adaptive policy achieves its primary goal of reducing conflicting transitions. The reduction is substantial for high-conflict programs: 43–98% for hsqldb6, xalan6, avrora9, pmd9, xalan9, and pjbb2005. Hybrid tracking provides little or no improvement for low-conflict programs—but they incur low coordination costs anyway.

The *Same state* column measures the downside of transitioning to pessimistic states: some transitions that *would* have been optimistic same-state become pessimistic. Only a small fraction of same-state transitions become pessimistic, because the adaptive policy identifies pessimistic objects to transition back to optimistic states, based on accurate profiling of pessimistic objects.

As the table shows, the adaptive policy causes more same-state than conflicting transitions to become pessimistic (compared with optimistic tracking alone). However, this result does not imply a performance loss, since a conflicting transition costs 2–3 orders of magnitude more than a same-state transition. For these programs at least, the adaptive policy achieves its goal of eliminating most of the conflicting transitions—and thus most of the expensive coordination overhead—while minimizing pessimistic transitions.

The *Pessimistic* columns show the number of pessimistic transitions under hybrid tracking. We note that deferred unlocking enables a significant fraction of *Uncontended* accesses to be *Reentrant* and thus avoid atomic operations. Still, a substantial fraction of pessimistic accesses require atomic operations, so pessimistic tracking alone would be costly even if it used deferred unlocking.

For most programs, a small fraction of pessimistic accesses are *Contended*, indicating that deferring unlocking of pessimistic states is generally successful. However, for avrora9 and pjbb2005, contended transitions are of the same order as optimistic conflicting transitions, so hybrid tracking still incurs a considerable amount of coordination. Investigating further, we find that the contention is, as expected, due to object-level data races. In pjbb2005, contention is caused by true (precise) data races. In avrora9, contention is caused by both true and false (object-level-only) data races.

The last two columns show transitions between pessimistic and optimistic states. Not all of the objects that transition from optimistic to pessimistic should ideally be pessimistic. The fraction of pessimistic objects transitioned *back* to optimistic states varies significantly across the programs but is often substantial, indicating that accurate profiling of pessimistic objects is crucial.

7.5 Performance of Tracking Alone

Figure 7 compares the performance of hybrid tracking with pessimistic and optimistic tracking alone (no runtime support on top of dependence tracking). Each bar shows the run-time overhead added over unmodified Jikes RVM. For sunflow9, the mean overhead is noticeably higher than the median for several configurations. Across many additional trials, we found that about 15% of the trials run substantially slower than the rest of the trials.

Pessimistic tracking adds 340% overhead on average (excluding sunflow9, the geomean is 210%), showing that pessimistic states must be applied judiciously. In contrast, the average overhead of *Optimistic tracking* is just 28%, but a few high-conflict programs (xalan6 and pjbb2005) incur substantially higher costs.

Hybrid tracking w/infinite cutoff uses hybrid tracking but sets $Cutoff_{confl}$ to ∞, so no object ever transitions to pessimistic states. This configuration measures only the costs, not the benefits, of hybrid tracking over optimistic tracking. The average cost over optimistic tracking is 2.3% (of baseline execution time).

Hybrid tracking uses the default values of $Cutoff_{confl}$ and other parameters. Hybrid tracking significantly improves the performance of several programs that perform poorly with optimistic tracking—the same programs that have many conflicting transitions reduced by the adaptive policy (Table 2). Hybrid tracking reduces overhead by 63% (65% $\rightarrow$ 24%) for xalan6; by 74% (19% $\rightarrow$ 5%) for xalan9; and by 45% (from 110% $\rightarrow$ 49%) for pjbb2005. Despite reducing conflicting transitions significantly for hsqldb6 (Table 2), hybrid tracking has little performance impact because hsqldb6's conflicting transitions mainly use implicit coordination, which costs about as much as a pessimistic transition.

Ideal is the overhead of optimistic tracking, but *without performing coordination* for conflicting transitions. This *unsound* configuration estimates the cost of all conflicting transitions becoming pessimistic and all same-state transitions remaining optimistic. It adds 14% on average, representing an estimated upper bound on the performance that hybrid tracking might be able to provide.

Hybrid tracking adds 22% average overhead, 21% less than optimistic tracking's 28% overhead. Hybrid tracking incurs 27% less overhead than *Hybrid tracking w/infinite cutoff*, recovering most of the overhead difference between optimistic tracking alone and the ideal, unsound configuration.

While optimistic tracking provides the best performance for low-conflict programs, hybrid tracking provides better performance for high-conflict programs. On average, hybrid tracking adds lower overhead than both pessimistic and optimistic tracking alone.

Many of the programs we evaluate perform relatively little shared-memory communication [21]. These programs may or may

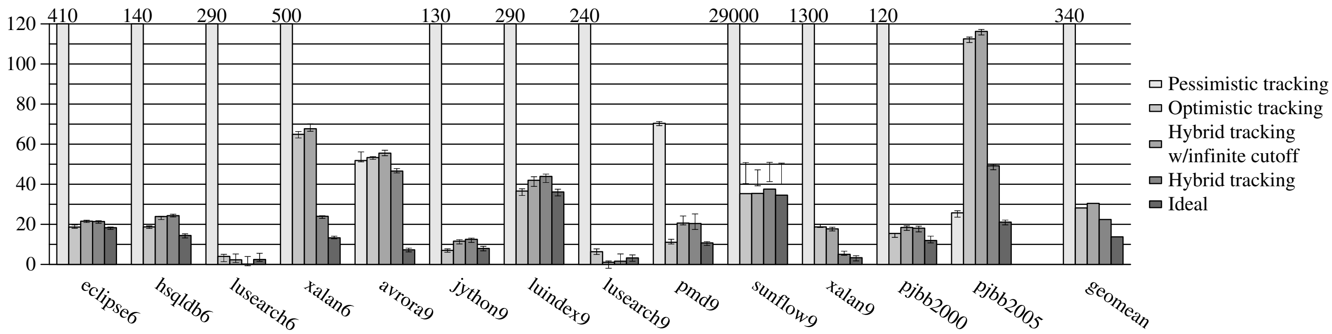
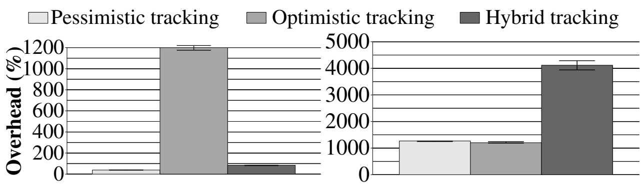


Figure 7. Run-time overhead of pessimistic and optimistic tracking, compared with hybrid tracking. Each bar is the median of 20 trials. The intervals are 95% confidence intervals centered at the mean. Overheads exceeding 120% are labeled using two significant figures.



(a) syncInc (b) racyInc

```
for (int i=0; i<MAX; i++) {
  synchronized (gLock) {
    gCounter++;
  }
}
```

```
for (int i=0; i<MAX; i++) {
  gCounter++;
}
```

Figure 8. Run-time overhead of tracking alone on microbenchmarks.

not accurately represent all real-world parallel programs in the wild. Because of these programs' low average communication, optimistic tracking performs well on average, leaving little room for hybrid tracking to improve. Nevertheless, only hybrid tracking can scale to diverse communication patterns: it helps cases for which optimistic tracking performs poorly, without harming cases for which optimistic tracking performs well.

Stress tests. In addition to large, real programs, we evaluate pessimistic, optimistic, and hybrid tracking on two microbenchmarks—one well synchronized and one with data races—that represent extreme, high-conflict cases. Each microbenchmark spawns eight threads; each thread repeatedly increments a global counter in a loop. Figure 8 shows, for each microbenchmark, the code executed by each thread, as well as run-time overhead over execution time on the unmodified JVM. The program syncInc acquires a global lock before every increment, whereas racyInc does not.

The figure shows that for syncInc, hybrid tracking significantly reduces overhead relative to optimistic tracking (84% versus 1200%), eliminating most coordination thanks to object-level data race freedom. For this program, hybrid tracking essentially mimics pessimistic tracking by using pessimistic transitions. However, hybrid tracking incurs more overhead in order to defer unlocking states and to perform profiling.

In contrast, racyInc represents a worst case for hybrid tracking since almost all conflicting accesses are involved in data races. Hybrid tracking adds 4300% overhead because threads repeatedly trigger coordination in order to perform pessimistic contended transitions. Upon further investigation, we find that although only 24% of memory accesses perform pessimistic contended transitions, most of these accesses trigger coordination more than once. Hybrid tracking could alleviate this deficiency by modifying the adaptive policy to switch a pessimistic object back to optimistic states if accesses to it trigger coordination frequently.

Pessimistic and optimistic tracking both add about 1200% overhead for racyInc; this similarity is initially surprising considering that racyInc executes many conflicting accesses, which are typically more expensive for optimistic tracking than for pessimistic tracking. We find that in optimistic tracking, only 8.5% of all accesses trigger conflicting transitions, because a thread that locks a state can perform several same-state transitions before another thread initiates a conflicting transition. In contrast, in pessimistic tracking, another thread tries to lock a state more quickly, leading to more remote cache misses: 26% of pessimistic tracking's accesses lock a state with a different thread than the previous access.

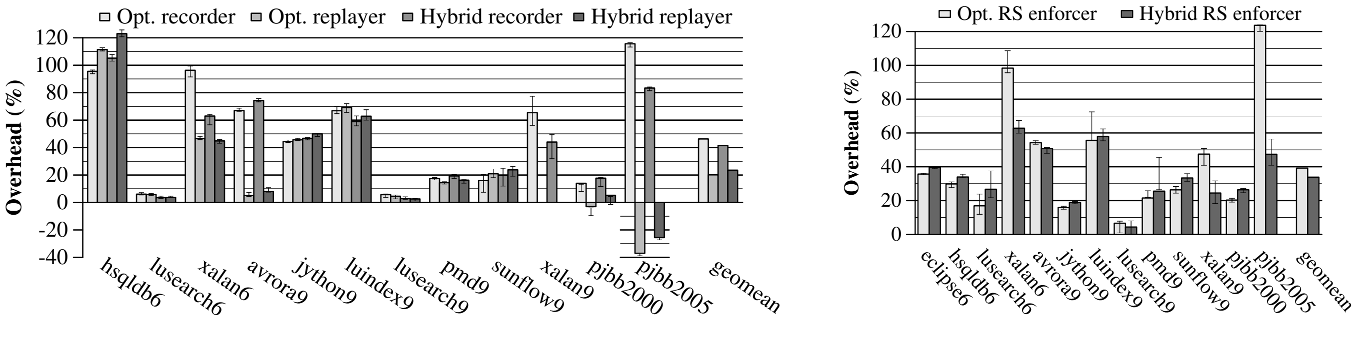
7.6 Performance of Runtime Support

This section compares optimistic and hybrid versions of the dependence recorder and RS enforcer. We have not implemented or evaluated *pessimistic* runtime support, since pessimistic tracking *alone* is slower than both optimistic and hybrid runtime support.

Dependence recorder. Figure 9(a) shows the performance of the optimistic and hybrid dependence recorders and replayers. Hybrid tracking improves the recorder's performance significantly for the high-conflict programs xalan6, xalan9, and pjbb2005, and incurs modest overhead for low-conflict programs. On average it reduces overhead by 11% (from 46 to 41%). While the hybrid recorder triggers less coordination than the optimistic recorder, it still detects and records the same number of cross-thread dependences as the optimistic recorder does. This fact explains why the hybrid recorder's improvement over the optimistic recorder is smaller than for hybrid tracking over optimistic tracking alone.

The optimistic *replayer* is not fully robust: it successfully replays 11 out of 13 programs (failing on eclipse6 and xalan9) [10]. The optimistic replayer adds 20% overhead on average—lower than the optimistic recorder because it is cheaper to replay known dependences than record unknown dependences. The replayer outperforms the *baseline* substantially for pjbb2005. This result is *not* an experimental anomaly; the replayer elides program synchronization operations and replays only the recorded dependences, so it can outperform baseline execution for programs dominated by coarse-grained, overly conservative synchronization.

Our hybrid replayer successfully replays all 11 programs that the optimistic replayer can replay. The hybrid replayer adds 24% overhead on average, slower than the optimistic replayer, due to the cost of maintaining the per-thread release counter, as well as the fact that hybrid tracking cannot reduce the number of replayed cross-thread dependences. Overall, hybrid tracking im-



(a) Dependence recorders and replayers

(b) Region serializability enforcers

Figure 9. Run-time overhead of optimistic and hybrid runtime support.

proves record time and degrades replay time—a worthwhile tradeoff since (1) optimizing record is more important since it is usually slower than replay, and (2) replay performance is not important in all settings (e.g., offline replay).

Region serializability enforcer. Figure 9(b) shows the overhead of enforcing SBRS using optimistic versus hybrid tracking. The hybrid enforcer substantially improves the performance of xalan6, xalan9, and pjbb2005. This reduction is similar to the reduction between hybrid and optimistic tracking alone—which is unsurprising since the hybrid enforcer employs hybrid tracking in essentially the same way as the optimistic enforcer employs optimistic tracking. On average, the hybrid enforcer reduces overhead by 13% over the optimistic enforcer (from 39% to 34%).

The performance story for runtime support is similar to the story for dependence tracking alone: hybridizing pessimistic and optimistic tracking overcomes the limitations of both, providing the best overall performance for a mix of low- and high-conflict programs.

8. Related Work

This section compares with prior work not covered already.

Program locks. This paper focuses on locks that are used by runtime support and are not visible to programmers. *Program* locks face similar tradeoffs as pessimistic versus optimistic tracking. Notably, *biased locking* avoids atomic operations for repeated lock acquisitions by the same thread, requiring coordination when another thread acquires the lock [13, 22, 33]. A biased lock typically falls back to an unbiased lock after triggering coordination once.

Adaptive mechanisms. Prior work has used adaptive techniques to combine different kinds of synchronization. Usui et al. use online profiling and a cost–benefit model to adaptively choose between lock-based mutual exclusion and software transactional memory (STM) for enforcing atomicity of critical sections [37]. Abadi et al. present an STM that adaptively changes how it detects conflicts for non-transactional accesses, depending on whether transactions access the same objects as non-transactional code [1]. Dice et al. build a runtime library that supports adaptive lock elision using hardware transactional memory (HTM) and optimistic software execution [16]. Ziv et al. formalize a theory for correctly composing different concurrency control protocols in programs [43].

Tracking dependences using commodity hardware. Intel's recently introduced Haswell architecture provides *restricted transactional memory* (RTM): best-effort TM support with an upper bound on shared-memory accesses in a transaction [42]. Recent work finds that an RTM transaction must be expanded to replace at least 3–4 atomic operations, in order to amortize the overhead of a transaction [28, 31, 42]. While an empirical comparison with RTM is beyond this work's scope, prior results suggest that optimistic tracking is likely to outperform RTM for non-conflicting accesses by avoiding atomic operations altogether, while hybrid tracking is likely to perform best for a mix of high- and low-conflict accesses.

9. Conclusion

Hybrid tracking uses a hybrid state model and adaptive policy to combine pessimistic and optimistic tracking effectively and efficiently, achieving better overall performance than either alone. We demonstrate hybrid tracking's potential by building runtime support to record dependences and enforce region serializability. The results motivate hybrid tracking's use in building efficient runtime support that targets diverse applications on commodity systems.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Tim Harris, for valuable feedback on the text. Thanks to Swarnendu Biswas, Joe Devietti, Jipeng Huang, Milind Kulkarni, Olatunji Ruwase, Xipeng Shen, and Ben Wood for helpful discussions.

A. Instrumentation Pseudocode

Figure 10 shows the instrumentation added by hybrid tracking. For simplicity, we only show instrumentation for a program store. The instrumentation for loads is more complex because it handles $\mathsf{RdEx}_\mathsf{T}^*$ and $\mathsf{RdSh}_\mathsf{c}^*$ states and supporting reentrant reader locks.

The fast path (Figure 10(a)) only checks for the $\mathsf{WrEx}_\mathsf{T}^{\mathsf{Opt}}$ state, since we expect that the majority of accesses trigger same-state optimistic transitions. The slow path (Figure 10(b)) changes the state based on hybrid tracking's state transitions (Figure 3 and Table 3). The slow path repeatedly reloads and tries to change the state if an atomic update fails. A contended transition triggers coordination (line 39); then the slow path retries until the state becomes unlocked, enabling an uncontended transition (lines 33–37). Upon a successful transition to a pessimistic state, the instrumentation adds the object to the per-thread lock buffer (lines 35 and 48).

Figure 10(c) shows the instrumentation at each PSRO and responding safe point. The instrumentation flushes the current thread's lock buffer by unlocking each object in the buffer, potentially transferring the object to an optimistic state, according to the adaptive policy (Section 6). The pseudocode shows how to handle objects in $\mathsf{WrEx}_\mathsf{T}^{\mathsf{WLock}}$ state only, not other states.

```
if (o.state != WrEx_T^Opt) {
  slowPath(o);
}
o.f = ... ; // program store
```

(a) The instrumentation fast path for hybrid tracking.

```
slowPath(o) {
  while(true) {
    state = o.state;
    if (isPess(state)) { // Pessimistic
      // Pessimistic Locked, uncontended
      if (state == WrEx_T^WLock) break;
      // Pessimistic Unlocked
      if (isUnlocked(state) || state == WrEx_T^RLock) {
        if (CAS(&o.state, state, WrEx_T^WLock)) {
          T.lockBuffer.add(o);
          break;
        }
      } else { // Pessimistic Locked, contended
        coordinate(getOwner(state));
      }
    } else { // Optimistic
      if (state == RdEx_T^Opt) { ... }
      if ((state != Int_*) && CAS(&o.state, state, Int_T)) {
        coordinate(getOwner(state));
        // Decision from adaptive policy
        if (AdaptivePolicy.toPess(o)) {
          o.state = WrEx_T^WLock;
          T.lockBuffer.add(o);
        } else {
          o.state = WrEx_T^Opt;
        }
        break;
      }
    }
    checkAndRespondToRequests(); // non-blocking safe point
  }
}
```

(b) The instrumentation slow path for hybrid tracking.

```
for (o : T.lockBuffer) {
  o.state = AdaptivePolicy.toOpt(o) ? WrEx_T^Opt : WrEx_T^Pess;
}
```

(c) Instrumentation at PSROs and responding safe points.

Figure 10. Instrumentation added by hybrid tracking, for program stores only. (Handling loads is analogous but more complex.)

B. Complete State Transitions

Table 3 shows all possible transitions for the hybrid state model.[9] Rows above the double line are pessimistic transitions; rows below are optimistic transitions. The rows labeled *Pessimistic unlock* ***OR*** *Pess* → *Opt* show transitions for deferred unlocking, which occur at program synchronization release operations (PSRO).

Each thread keeps track of which objects it has read-locked in a per-thread *read set*, T.rdSet. The table omits the following details: When T reads an object not in its read set ($o \notin T.rdSet$), it adds the object to its read set: $T.rdSet \leftarrow T.rdSet \cup \{o\}$. Whenever T flushes its lock buffer, it also clears its read set: $T.rdSet \leftarrow \emptyset$.

[9] An early version of our work introduces a significantly different hybrid state model (e.g., it does *not* use deferred unlocking) and thus presents significantly different state transitions [14].

References

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional Memory with Strong Atomicity Using Off-the-Shelf Memory Protection Hardware. In *PPoPP*, pages 185–196, 2009.

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

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

[4] 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.

[5] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin Locks: Featherweight Synchronization for Java. In *PLDI*, pages 258–268, 1998.

[6] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia. Valor: Efficient, Software-only Region Conflict Exceptions. In *OOPSLA*, pages 241–259, 2015.

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

[8] H.-J. Boehm. Position paper: Nondeterminism is Unavoidable, but Data Races are Pure Evil. In *RACES*, pages 9–14, 2012.

[9] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, pages 68–78, 2008.

[10] M. D. Bond, M. Kulkarni, M. Cao, M. Fathi Salmi, and J. Huang. Efficient Deterministic Replay of Multithreaded Executions in a Managed Language Virtual Machine. In *PPPJ*, pages 90–101, 2015.

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

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

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

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

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

[16] D. Dice, A. Kogan, Y. Lev, T. Merrifield, and M. Moir. Adaptive Integration of Hardware and Software Lock Elision Techniques. In *SPAA*, pages 188–197, 2014.

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

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

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

[20] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *OOPSLA*, pages 388–402, 2003.

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

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

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

Trans. type	Old state	Program access	New state	Sync. needed	Cross-thread dependence?
Pessimistic uncontended (reentrant)	$\mathsf{WrEx}_\mathsf{T}^\mathsf{WLock}$	R or W by T	Same	None	No
	$\mathsf{WrEx}_\mathsf{T}^\mathsf{RLock}$	R by T			
	$\mathsf{RdEx}_\mathsf{T}^\mathsf{RLock}$	R by T			
	$\mathsf{RdSh}_\mathsf{c}^{\mathsf{RLock}(\mathsf{n})}$	R by T if $\mathsf{o} \in \mathsf{T.rdSet}$			
Pessimistic uncontended	$\mathsf{WrEx}_\mathsf{T}^\mathsf{Pess}$	W by T	$\mathsf{WrEx}_\mathsf{T}^\mathsf{WLock}$	CAS	No
	$\mathsf{WrEx}_\mathsf{T}^\mathsf{Pess}$	R by T	$\mathsf{WrEx}_\mathsf{T}^\mathsf{RLock}$		
	$\mathsf{RdEx}_\mathsf{T}^\mathsf{Pess}$	R by T	$\mathsf{RdEx}_\mathsf{T}^\mathsf{RLock}$		
	$\mathsf{RdEx}_\mathsf{T}^\mathsf{Pess}$ or $\mathsf{RdEx}_\mathsf{T}^\mathsf{RLock}$ or $\mathsf{WrEx}_\mathsf{T}^\mathsf{RLock}$	W by T	$\mathsf{WrEx}_\mathsf{T}^\mathsf{WLock}$	CAS	No
	$\mathsf{RdEx}_\mathsf{T1}^\mathsf{Pess}$	R by $\mathsf{T2}$	$\mathsf{RdSh}_\mathsf{gRdShCount}^{\mathsf{RLock}(1)}$		Maybe
	$\mathsf{RdEx}_\mathsf{T1}^\mathsf{RLock}$ or $\mathsf{WrEx}_\mathsf{T1}^\mathsf{RLock}$	R by $\mathsf{T2}$	$\mathsf{RdSh}_\mathsf{gRdShCount}^{\mathsf{RLock}(2)}$		Maybe
	$\mathsf{RdSh}_\mathsf{c}^\mathsf{Pess}$	R by T	$\mathsf{RdSh}_\mathsf{c}^{\mathsf{RLock}(1)*}$	CAS	Maybe
	$\mathsf{RdSh}_\mathsf{c}^{\mathsf{RLock}(\mathsf{n})}$	R by T if $\mathsf{o} \notin \mathsf{T.rdSet}$	$\mathsf{RdSh}_\mathsf{c}^{\mathsf{RLock}(\mathsf{n}+1)*}$		
	$\mathsf{WrEx}_\mathsf{T1}^\mathsf{Pess}$	W by $\mathsf{T2}$	$\mathsf{WrEx}_\mathsf{T2}^\mathsf{WLock}$	CAS	Maybe
	$\mathsf{WrEx}_\mathsf{T1}^\mathsf{Pess}$	R by $\mathsf{T2}$	$\mathsf{RdEx}_\mathsf{T2}^\mathsf{RLock}$		
	$\mathsf{RdEx}_\mathsf{T1}^\mathsf{Pess}$	W by $\mathsf{T2}$	$\mathsf{WrEx}_\mathsf{T2}^\mathsf{WLock}$		
	$\mathsf{RdSh}_\mathsf{c}^\mathsf{Pess}$	W by T	$\mathsf{WrEx}_\mathsf{T}^\mathsf{WLock}$		
Pessimistic contended	$\mathsf{WrEx}_\mathsf{T1}^\mathsf{WLock}$	W by $\mathsf{T2}$	Handled at owner thread(s)' responding safe points	Roundtrip coordination	Maybe
	$\mathsf{WrEx}_\mathsf{T1}^\mathsf{RLock}$	W by $\mathsf{T2}$			
	$\mathsf{WrEx}_\mathsf{T1}^\mathsf{WLock}$	R by $\mathsf{T2}$			
	$\mathsf{RdEx}_\mathsf{T1}^\mathsf{RLock}$	W by $\mathsf{T2}$			
	$\mathsf{RdSh}_\mathsf{c}^{\mathsf{RLock}(\mathsf{n})}$	W by $\mathsf{T2}$			
Pessimistic unlock **OR** Pess $\rightarrow$ Opt	$\mathsf{WrEx}_\mathsf{T}^\mathsf{WLock}$ or $\mathsf{WrEx}_\mathsf{T}^\mathsf{RLock}$	PSRO or responding safe point	$\mathsf{WrEx}_\mathsf{T}^\mathsf{Pess}$ **OR** $\mathsf{WrEx}_\mathsf{T}^\mathsf{Opt}$	CAS	N/A
	$\mathsf{RdEx}_\mathsf{T}^\mathsf{RLock}$		$\mathsf{RdEx}_\mathsf{T}^\mathsf{Pess}$ **OR** $\mathsf{RdEx}_\mathsf{T}^\mathsf{Opt}$		
	$\mathsf{RdSh}_\mathsf{c}^{\mathsf{RLock}(\mathsf{n})}$ if $\mathsf{n} > 1$		$\mathsf{RdSh}_\mathsf{c}^{\mathsf{RLock}(\mathsf{n}-1)}$		
	$\mathsf{RdSh}_\mathsf{c}^{\mathsf{RLock}(1)}$		$\mathsf{RdSh}_\mathsf{c}^\mathsf{Pess}$ **OR** $\mathsf{RdSh}_\mathsf{c}^\mathsf{Opt}$		
Same state	$\mathsf{WrEx}_\mathsf{T}^\mathsf{Opt}$	R or W by T	Same	None	No
	$\mathsf{RdEx}_\mathsf{T}^\mathsf{Opt}$	R by T			
	$\mathsf{RdSh}_\mathsf{c}^\mathsf{Opt}$	R by T if $\mathsf{T.rdShCount} \geq \mathsf{c}$			
Upgrading	$\mathsf{RdEx}_\mathsf{T}^\mathsf{Opt}$	W by T	$\mathsf{WrEx}_\mathsf{T}^\mathsf{Opt}$	CAS	No
	$\mathsf{RdEx}_\mathsf{T1}^\mathsf{Opt}$	R by $\mathsf{T2}$	$\mathsf{RdSh}_\mathsf{gRdShCount}^\mathsf{Opt}$		Maybe
Fence	$\mathsf{RdSh}_\mathsf{c}^\mathsf{Opt}$	R by T if $\mathsf{T.rdShCount} < \mathsf{c}$	$(\mathsf{T.rdShCount} \leftarrow \mathsf{c})$	Memory fence	Maybe
Conflicting **OR** Opt $\rightarrow$ Pess	$\mathsf{WrEx}_\mathsf{T1}^\mathsf{Opt}$	W by $\mathsf{T2}$	$\mathsf{Int}_\mathsf{T2} \rightarrow \mathsf{WrEx}_\mathsf{T2}^\mathsf{Opt}$ **OR** $\mathsf{WrEx}_\mathsf{T2}^\mathsf{WLock}$	Roundtrip coordination	Maybe
	$\mathsf{WrEx}_\mathsf{T1}^\mathsf{Opt}$	R by $\mathsf{T2}$	$\mathsf{Int}_\mathsf{T2} \rightarrow \mathsf{RdEx}_\mathsf{T2}^\mathsf{Opt}$ **OR** $\mathsf{RdEx}_\mathsf{T2}^\mathsf{RLock}$		
	$\mathsf{RdEx}_\mathsf{T1}^\mathsf{Opt}$	W by $\mathsf{T2}$	$\mathsf{Int}_\mathsf{T2} \rightarrow \mathsf{WrEx}_\mathsf{T2}^\mathsf{Opt}$ **OR** $\mathsf{WrEx}_\mathsf{T2}^\mathsf{WLock}$		
	$\mathsf{RdSh}_\mathsf{c}^\mathsf{Opt}$	W by $\mathsf{T2}$	$\mathsf{Int}_\mathsf{T2} \rightarrow \mathsf{WrEx}_\mathsf{T2}^\mathsf{Opt}$ **OR** $\mathsf{WrEx}_\mathsf{T2}^\mathsf{WLock}$		

Table 3. All possible state transitions for the hybrid state model. Instances of "**OR**" indicate cases in which a state can potentially transition between pessimistic and optimistic states. * Pessimistic uncontended transitions from $\mathsf{RdSh}_\mathsf{c}^*$ to $\mathsf{RdSh}_\mathsf{c}^{\mathsf{RLock}(*)}$ also update $\mathsf{T.rdShCount}$ to $\max(\mathsf{T.rdShCount}, \mathsf{c})$.

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

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

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

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

[28] H. S. Matar, I. Kuru, S. Tasiran, and R. Dementiev. Accelerating Precise Race Detection Using Commercially-Available Hardware Transactional Memory Support. In *WoDet*, 2014.

[29] J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. ...and region serializability for all. In *HotPar*, 2013.

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

[31] C. G. Ritson and F. R. Barnes. An Evaluation of Intel's Restricted Transactional Memory for CPAs. In *CPA*, pages 271–292, 2013.

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

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

[34] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In *PPoPP*, pages 187–197, 2006.

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

[36] A. Sengupta, S. Biswas, M. Zhang, M. D. Bond, and M. Kulkarni. Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability. In *ASPLOS*, pages 561–575, 2015.

[37] T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In *PACT*, pages 3–14, 2009.

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

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

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

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

[42] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing. In *SC*, pages 19:1–19:11, 2013.

[43] O. Ziv, A. Aiken, G. Golan-Gueta, G. Ramalingam, and M. Sagiv. Composing Concurrency Control. In *PLDI*, pages 240–249, 2015.