Hybridizing and Relaxing Dependence Tracking for Efficient Parallel Runtime Support

MAN CAO, Ohio State University
MINJIA ZHANG, Microsoft Research
ARITRA SENGUPTA, Ohio State University
SWARNENDU BISWAS, University of Texas at Austin
MICHAEL D. BOND, Ohio State University

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

This article presents two novel approaches that seek to improve the performance of dependence tracking. *Hybrid tracking* (HT) hybridizes pessimistic and optimistic tracking by overcoming a fundamental mismatch between these two kinds of tracking. HT uses an adaptive, profile-based policy to make runtime decisions about switching between pessimistic and optimistic tracking. *Relaxed tracking* (RT) attempts to reduce optimistic tracking's overhead on conflicting accesses by tracking dependencies in a "relaxed" way—meaning that not all dependencies are tracked accurately—while still preserving both program semantics and runtime support's correctness. To demonstrate the usefulness and potential of HT and RT, we build runtime support based on the two approaches. Our evaluation shows that both approaches offer performance advantages over existing approaches, but there exist challenges and opportunities for further improvement.

HT and RT are distinct solutions to the same problem. It is easier to build runtime support based on HT than on RT, although RT does not incur the overhead of online profiling. This article presents the two approaches together to inform and inspire future designs for efficient parallel runtime support.

CCS Concepts: • **Computing methodologies → Parallel computing methodologies**; • **Software and its engineering → Runtime environments**; *Software performance*; *Software reliability*;

Additional Key Words and Phrases: Dynamic analysis, synchronization, concurrency correctness, runtime support for parallelism, dependence tracking, data races

This work is supported by the National Science Foundation under Grants CSR-1218695, CAREER-1253703, CCF-1421612, and XPS-1629126. This journal article synthesizes work previously published in Cao et al. (2016) and Zhang et al. (2016).
Authors' addresses: M. Cao, A. Sengupta, and M. D. Bond, Department of Computer Science and Engineering, Ohio State University, 2015 Neil Avenue, 395 Dreese Labs, Columbus, OH 43210; emails: {caoma, sengupta, mikebond}@cse.ohio-state.edu; M. Zhang, Microsoft Research, 14820 NE 36th St, Building 99, Redmond, WA 98052; email: minjiaz@microsoft.com; S. Biswas, Institute for Computational Engineering and Sciences, University of Texas at Austin, 201 E 24th St, POB 4.124, Austin, TX 78712; email: sbiswas@ices.utexas.edu.

https://doi.org/10.1145/3108138

ACM Reference format:
Man Cao, Minjia Zhang, Aritra Sengupta, Swarnendu Biswas, and Michael D. Bond. 2017. Hybridizing and Relaxing Dependence Tracking for Efficient Parallel Runtime Support. *ACM Trans. Parallel Comput.* 4, 2, Article 9 (August 2017), 42 pages.
https://doi.org/10.1145/3108138

1 INTRODUCTION

Parallel programs are becoming increasingly popular to make software scale with successive microprocessor generations that provide more, instead of faster, cores. However, it is notoriously difficult to achieve both correctness and scalability. In *shared-memory* parallel programs, *locks* offer an easy-to-understand concurrency control mechanism, but their use often leads to concurrency bugs and scalability bottlenecks.

Researchers have developed dynamic program analyses and software systems that help support reliable, scalable shared-memory parallelism. This article 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., Flanagan and Freund (2009)), software transactional memory (e.g., Harris and Fraser (2003)), enforcement of strong memory models (e.g., Ouyang et al. (2013)), atomicity checkers (e.g., Flanagan et al. (2008)), and multithreaded record and replay (e.g., Veeraraghavan et al. (2011)). 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 to *track* (detect or control) *cross-thread dependencies* (data dependencies involving two threads). This instrumentation is particularly costly, because it must add its own synchronization 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., Flanagan and Freund (2009), LeBlanc and Mellor-Crummey (1987), Lee et al. (2012), Harris and Fraser (2003), and Flanagan et al. (2008)), which we refer to as *pessimistic tracking* of dependencies. The performance of runtime support built on pessimistic tracking is relatively insensitive to the number of cross-thread dependencies 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 dependencies but incurs significant latency at conflicting accesses to perform coordination between threads (Russell and Detlefs 2006; Kawachiya et al. 2002; Burrows 2004; Scales et al. 1996; von Praun and Gross 2001; Bond et al. 2013). 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 to build high-performance runtime support.

This article proposes two novel, distinct approaches that aim to overcome the limitations of both pessimistic and optimistic tracking. The first approach, called *hybrid tracking* (HT), combines pessimistic and optimistic tracking to benefit from both. HT addresses a fundamental mismatch between pessimistic and optimistic tracking, introducing a novel approach that defers unlocking of pessimistic states, based on insights about the interplay between dependence tracking and program synchronization. HT consists of two components: a *hybrid state model* that supports shared variables being in—and transferring between—pessimistic and optimistic *states* (i.e., handled by pessimistic and optimistic tracking, respectively) and an *adaptive policy* that makes profile-guided decisions about when to apply pessimistic versus optimistic tracking.

The second approach, called *relaxed tracking* (RT), seeks to hide the latency of optimistic tracking's coordination at conflicting accesses. To do so, RT relaxes the requirement that runtime support must track all dependencies accurately—while preserving the runtime support's guarantees and adhering to the language's semantics. RT enables a thread to continue executing past a memory access involved in a cross-thread dependence, without accurately tracking the dependence. RT's design consists of two components: a *relaxed coordination protocol* and support for *relaxed loads and stores*.

We extend existing runtime support to use HT and RT to demonstrate their usefulness. We build two dependence recorders based on HT and RT, respectively. In addition, we build runtime support for enforcement of region serializability based on HT, and a software transactional memory (STM) system based on RT. We note that although the two techniques are potentially complementary, we have not combined them or evaluated them together, due to the complexity and challenges of doing so.

We have implemented HT, RT, and the hybrid and relaxed runtime support in a Java virtual machine that performs competitively with commerical JVMs (Biswas et al. 2015). Our evaluation shows that although both HT and RT's *average* performance improvement over optimistic tracking is modest, they (1) outperform pessimistic tracking consistently, (2) outperform optimistic tracking for several high-conflict programs, and (3) perform about the same as optimistic tracking for low-conflict programs.

Compared with RT, using HT requires fewer modifications to runtime support. On the other hand, HT's adaptive policy relies on online profiling and incurs slightly higher overhead than RT for low-conflict programs. Much of HT's performance benefit directly translates into speedup of hybrid runtime support. In contrast, RT's potential for improving performance is often limited by correctness contraints and by RT's indirect effects on dependence tracking behavior.

Overall, these results demonstrate the potential for using novel mechanisms to address a key performance bottleneck of parallel runtime support, suggesting new directions for efficient, flexible, software-only runtime support that targets a variety of parallel software systems.

2 BACKGROUND AND MOTIVATION

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

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

For data-race-free (DRF) programs, runtime support can track cross-thread dependencies soundly by instrumenting only program synchronization operations, because shared-memory languages such as Java and C++ guarantee serializability of synchronization-free regions for DRF programs (Boehm and Adve 2008; Manson et al. 2005; Adve and Hill 1990; Adve and Boehm 2010). However, programs routinely have data races, which are hard to detect or eliminate (e.g., Flanagan and Freund (2009), von Praun and Gross (2003), Lee et al. (2012), and Boyapati et al. (2002)), 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 (von Praun and Gross 2003; Lee et al. 2012; Choi et al. 2002; Elmas et al. 2007).)

Table 1. All Possible State Transitions for Last-Access States

Transition type	**Old state**	**Access**	**New state**	**Synchronization required** Pessimistic tracking	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
	RdEx_{T1}	R by T2	RdSh_c*		
Fence	RdSh_c	R by T	Same*	CAS	Memory fence
Conflicting	WrEx_{T1}	W by T2	WrEx_{T2}	CAS	Coordination
	WrEx_{T1}	R by T2	RdEx_{T2}		
	RdEx_{T1}	W by T2	WrEx_{T2}		
	RdSh_c	W by T	WrEx_T		

*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 T.rdShCount < c [Bond et al. 2013].

Tracking cross-thread dependencies. To track cross-thread dependencies, 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 dependencies.[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 (Lamport 1978) that transitively imply all of an execution's cross-thread dependencies (Bond et al. 2013).

Same-state transitions involve no state change; they do not imply any cross-thread dependencies. Other transitions imply potential cross-thread dependencies. *Upgrading* transitions either transitively indicate write–read dependencies or help detect later write–write dependencies. *Fence* transitions enable detecting write–read dependencies when a thread reads a RdSh_c object for the first time (prior work provides details (Bond et al. 2013), which are not integral to understanding this article). Finally, *conflicting* transitions directly indicate write–write, write–read, or read–write dependencies.

Instrumentation atomicity. To track dependencies 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 dependencies; 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 dynamic data race detection, which requires only instrumentation atomicity, because it does not need to

[1]This article 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 dependencies (Bond et al. 2013).

know the order of racy accesses.) In any case, instrumentation atomicity and instrumentation–access atomicity incur similar costs.

To guarantee instrumentation–access atomicity in the presence of data races, much 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 used by runtime support only, and are not visible 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), then a potential cross-thread dependence exists, requiring additional runtime-support-specific work (indicated by the comment */* handle ... */*). For example, a dependence recorder might record the dependence in a log; STM might check whether the access conflicts with an ongoing transaction (and pause or abort a transaction if so).

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 4× on average (Section 6.2.2).

Existing runtime support commonly employs pessimistic tracking (e.g., Flanagan and Freund (2009), LeBlanc and Mellor-Crummey (1987), Lee et al. (2012), Harris and Fraser (2003), and Flanagan et al. (2008)). We note that existing approaches often avoid performing an atomic operation for every memory access. For example, STM systems can use instrumentation that avoids atomic operations for accesses to the same object in the same transaction (Harris and Fraser 2003). STM can avoid atomic operations at loads by validating them lazily, which requires memory fences (e.g., Saha et al. (2006)), although many of these can be safely removed (Dalessandro and

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

```
slowPath(o) {
  state = o.state;
  if (state == $RdEx_T$) {
    ... ; // upgrading transition to $WrEx_T$
    return;
  }
  // Coordination for conflicting transition:
  while (state == $WrEx_*^{Int}$ || state == $RdEx_*^{Int}$
        || !CAS(&o.state, state, $WrEx_T^{Int}$)) {
    /* blocking safe point */
    state = o.state;  // re-read state
  }
  coordinate(getOwner(state));
  o.state = $WrEx_T$;
}

coordinate(remoteT) {
  response = sendRequest(remoteT); // return
      true if implicit coordination used
  while (!response) {
    /* blocking safe point */
    response = checkResponse(remoteT);
  }
}
```

Fig. 1. Pseudocode for optimistic tracking's instrumentation slow path (for program stores only) and coordination. T is the executing thread. The pseudocode omits memory fences required by the implementation.

Scott 2012), and stores and associated synchronization can be deferred until commit and uses less synchronization but potentially hurt scalability (e.g., Dalessandro et al. (2010)). Data race detectors can avoid atomic operations for repeated accesses in the same synchronization-free region (e.g., Flanagan and Freund (2009)). 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, for example, DoublePlay detects conflicts implicitly by using speculation and replication, but it adds high overhead unless extra cores are available (Veeraraghavan et al. 2011).

2.2 Optimistic Tracking

In contrast, optimistic tracking avoids synchronization at most accesses. Prior work uses optimistic tracking either to implement program locks (also known as *biased locking*) (Section 7) (Russell and Detlefs 2006; Kawachiya et al. 2002; Burrows 2004) or to track cross-thread dependencies (von Praun and Gross 2001; Bond et al. 2013; Scales et al. 1996; Jiang et al. 2015). This article 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); /* handle potential cross−thread dependence(s) */
}
o.f = ...; // program store
```

If the object's state is already $WrEx_T$, then the instrumentation takes only 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 dependencies.

Conflicting transitions require coordination. In optimistic tracking, conflicting transitions (last four rows of Table 1) require that threads *coordinate* with each other 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. Suppose a thread, which we call the *requesting thread*, reqT, wants to write to an object that was last accessed by other thread(s), each of which we call a *responding thread*, respT. If the object's state is $\mathsf{WrEx}_{\mathsf{respT}}$ or $\mathsf{RdEx}_{\mathsf{respT}}$, then there is one responding thread, respT. If its state is RdSh_{c}, then all other threads are responding threads, and reqT coordinates with each responding thread separately. For simplicity of exposition, we describe the case of a single responding thread respT. To initiate coordination, reqT first atomically changes the object's state to an *intermediate* state $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{reqT}}$ (line 9), which simplifies the protocol by allowing only one thread at a time to initiate coordination for an object. If there is another thread that has already changed the object to an intermediate state, then reqT waits for the other thread to finish coordination (lines 8–12). reqT then coordinates with respT (line 13) to ensure that reqT's state change does not interrupt respT's instrumentation–access atomicity.

The responding thread respT participates in coordination only when it is at a *safe point*: a program point that is definitely *not* in the middle of instrumentation or its corresponding access—thus preserving instrumentation–access atomicity. 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, for example, for stop-the-world garbage collection. Blocking operations, such as waiting to acquire a lock or for I/O, are also safe points.

If respT is at a blocking safe point, then reqT coordinates with respT *implicitly* by updating respT's status atomically, which respT will see when it finishes blocking. The helper method sendRequest() returns true if and only if it performs an implicit request. Otherwise, reqT coordinates with respT *explicitly*: reqT sends a request to remoteT by adding a request to respT's *request queue* and then waits for respT to respond at respT's next safe point. (Figure 1 does *not* show the actions of respT.) 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 reqT waits for an explicit coordination response, it acts as a blocking safe point (line 19), so other threads trying to access other objects can perform implicit requests with reqT acting as a *responding* thread, thus avoiding deadlock. Figure 2 illustrates how coordination works when using an explicit request. Finally, reqT changes the state to $\mathsf{WrEx}_{\mathsf{T}}$ (line 14) and proceeds with its access.

Performance. Optimistic tracking exploits a tradeoff: It avoids synchronization in the common, non-conflicting case but requires coordination in the uncommon, conflicting case. For programs that perform little communication, optimistic tracking incurs low overhead, as Section 6.2.2 shows. However, for programs that perform more communication (e.g., as little as 0.5% of accesses conflicting), optimistic tracking incurs high overhead (e.g., >100% runtime overhead).

The following table reports costs of different kinds of state transitions, averaged across all programs (Section 6.1 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

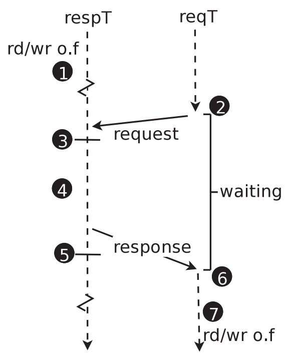


Fig. 2. Coordination using an explicit request. (1) respT accessed an object o previously. (2) reqT wants to access o. It changes o's state to $\text{RdEx}^{\text{Int}}_{\text{reqT}}$ or $\text{WrEx}^{\text{Int}}_{\text{reqT}}$ and enters a blocked state, waiting for respT's response. (3) respT reaches a safe point. (4) respT performs runtime-support-specific actions and then responds. (5) respT leaves the safe point. (6) reqT sees the response. (7) reqT changes o's state to $\text{WrEx}_{\text{reqT}}$ or $\text{RdEx}_{\text{reqT}}$ and proceeds to access o.

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.

Goals and outline. The limitations of pessimistic and optimistic tracking motivate our work in two directions:

—To develop a hybrid of pessimistic and optimistic tracking that uses optimistic tracking for most accesses but avoids most coordination by using pessimistic tracking for most conflicting accesses.
—To reduce or hide the coordination latency of explicit requests to improve optimistic tracking's performance.

Section 3 introduces an approach called *hybrid tracking* that combines pessimistic and optimistic tracking and extends existing runtime support based on this approach. Section 4 presents an approach called *relaxed tracking* that optimizes optimistic tracking's coordination that uses explicit requests and extends existing runtime support. The remaining sections describe our implementation and evaluation and compare with related work.

3 HYBRID TRACKING AND RUNTIME SUPPORT

This section presents HT, which combines pessimistic and optimistic tracking soundly and efficiently. Section 3.1 presents challenges inherent in combining pessimistic and optimistic tracking and introduces a hybrid state model that addresses these challenges. Sections 3.2 and 3.3 design sound and efficient runtime support using the hybrid state model. Section 3.4 describes a policy that decides between pessimistic and optimistic states at runtime.

3.1 Hybrid State Model

Hybridizing pessimistic and optimistic tracking is inherently difficult because of an inherent mismatch between them. The hybrid state model addresses this mismatch.

3.1.1 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, for a hybrid approach, 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 dependencies. This problem becomes clearer in the context of specific kinds of runtime support; Sections 3.2 and 3.3 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 (Cao et al. 2014). 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.

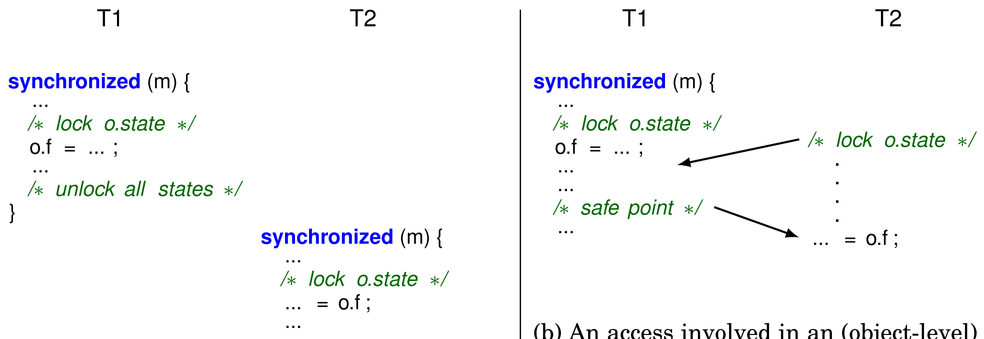
To overcome the mismatch between pessimistic and optimistic tracking that impaired our initial design, we introduce the following insight: The hybrid state model can and should *defer unlocking* of pessimistic states. Deferred unlocking consists of the following design points:

—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 deferred unlocking and 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, then 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 (von Praun and Gross 2001). The performance of deferred unlocking relies on object-level data races being rare, so few (if any) pessimistic transitions encounter contention.

Deferred unlocking is the key technical insight of HT. Intuitively, deferred 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 3 illustrates deferred unlocking of pessimistic states. The example assumes o is in pessimistic states for the accesses shown. In Figure 3(a), each thread executes a critical section acquiring the same program lock m. Code comments (e.g., */* lock o.state */*) summarize the runtime behavior of HT'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.



```
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 which case hybrid tracking triggers coordination.

Fig. 3. Deferred unlocking encounters contention only for object-level data races. Comments show instrumentation actions assuming o is in pessimistic states.

In contrast, in Figure 3(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 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.

3.1.2 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 $\text{WrEx}_{\text{T}}^{\text{Pess}}$, $\text{RdEx}_{\text{T}}^{\text{Pess}}$, and $\text{RdSh}_{\text{c}}^{\text{Pess}}$. The *pessimistic locked* states are $\text{WrEx}_{\text{T}}^{\text{RLock}}$, $\text{WrEx}_{\text{T}}^{\text{WLock}}$, $\text{RdEx}_{\text{T}}^{\text{RLock}}$, and $\text{RdSh}_{\text{c}}^{\text{RLock(n)}}$. To support reader–writer locking, a WrEx_{T} state can be either read- or write-locked, and a $\text{RdSh}_{\text{c}}^{\text{RLock(n)}}$ state is read-locked by n threads. The read-locked write-exclusive state ($\text{WrEx}_{\text{T}}^{\text{RLock}}$) enables a second concurrent reader to upgrade to $\text{RdSh}_{\text{c}}^{\text{RLock(2)}}$, instead of encountering contention. To support reentrant read locks, each thread also maintains a *read set* for objects whose states it has read-locked.
—The optimistic states are $\text{WrEx}_{\text{T}}^{\text{Opt}}$, $\text{RdEx}_{\text{T}}^{\text{Opt}}$, and $\text{RdSh}_{\text{c}}^{\text{Opt}}$.

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 4 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 4, focusing on transitions that are different from those shown in Table 1. Appendix A shows pseudocode for HT'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 4). For example, a read (or write) by T1 to an object in $\text{WrEx}_{\text{T1}}^{\text{Pess}}$ state triggers an uncontended transition to $\text{WrEx}_{\text{T1}}^{\text{RLock}}$ ($\text{WrEx}_{\text{T1}}^{\text{WLock}}$). A read by T2 to an object in $\text{WrEx}_{\text{T1}}^{\text{Pess}}$ triggers an uncontended transition to $\text{RdEx}_{\text{T2}}^{\text{RLock}}$.

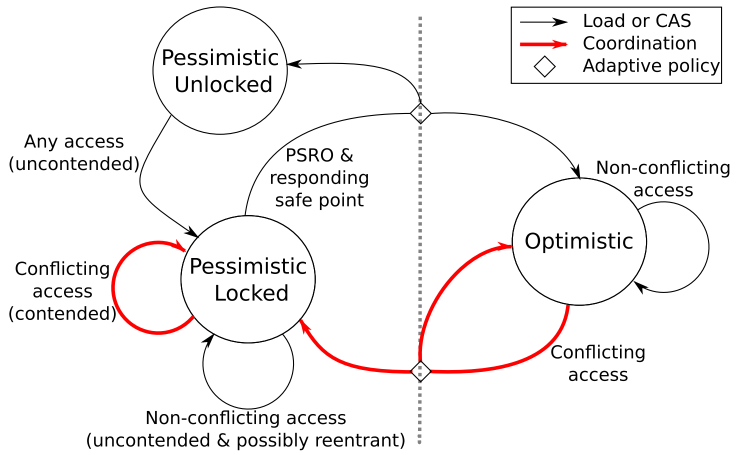


Fig. 4. High-level state transition diagram for the hybrid state model. The left and right halves show transitions starting in pessimistic and optimistic states, respectively. The diamonds on the vertical dashed line indicate decisions by the adaptive policy, described in Section 3.4.

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

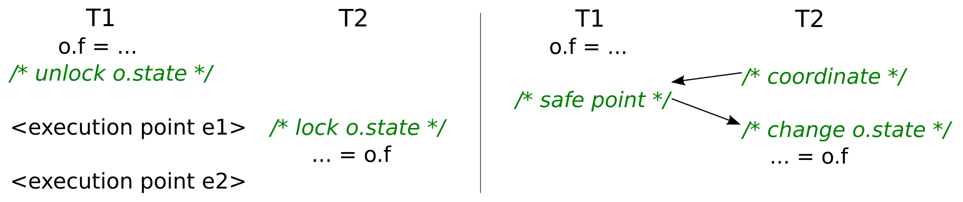
Unlocking of pessimistic states. To support deferred unlocking, each thread records every pessimistic object whose state it has locked in the thread's *lock buffer*. The lock buffer is logically a set of objects, but it can be implemented as a list of unique objects. A thread only needs to add an object to its lock buffer when it changes the object from a pessimistic unlocked state to a pessimistic locked state or when it read-locks a $\mathsf{RdSh}_{\mathsf{c}}^{\mathsf{RLock(n)}}$ object for the first time according to the read set, guaranteeing the uniqueness of objects in the lock buffer.

Every PSRO and responding safe point *flushes* the lock buffer by unlocking the states of all objects in the buffer (transition labeled "PSRO & responding safe point"). The objects can be unlocked in any order. Unlocking a $\mathsf{RdSh}_{\mathsf{c}}^{\mathsf{RLock(n)}}$ object means transitioning to $\mathsf{RdSh}_{\mathsf{c}}^{\mathsf{RLock(n-1)}}$ (if n>1) or the unlocked state $\mathsf{RdSh}_{\mathsf{c}}^{\mathsf{Pess}}$ (if n=1). Whenever a thread flushes its lock buffer, it also clears its read set for 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 3(b), a read by T2 to an object in $\mathsf{WrEx}_{\mathsf{T1}}^{\mathsf{WLock}}$ triggers a contended transition: T1 unlocks the state to $\mathsf{WrEx}_{\mathsf{T1}}^{\mathsf{Pess}}$ before responding to coordination. T2 then performs an uncontended transition from $\mathsf{WrEx}_{\mathsf{T1}}^{\mathsf{Pess}}$ to $\mathsf{RdEx}_{\mathsf{T2}}^{\mathsf{RLock}}$.



(a) pessimistic transition (b) optimistic transition

Fig. 5. The challenge of recording pessimistic transitions that involve conflicting states.

Transitions between pessimistic and optimistic states. The model supports transitioning to an optimistic state whenever it unlocks a pessimistic state (upper diamond in Figure 4) and to a pessimistic state from an optimistic state on any conflicting transition (lower diamond).

Although we have designed and presented HT 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 dependencies.

3.2 Recording and Replaying Dependencies

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

3.2.1 Optimistic Dependence Recorder and Replayer. Multithreaded record and replay helps programmers debug nondeterministic multithreaded programs, and it provides systems benefits such as replication-based fault tolerance (Ronsse and De Bosschere 1999; LeBlanc and Mellor-Crummey 1987; Lee et al. 2010, 2012; Weeratunge et al. 2010; Park et al. 2009; Veeraraghavan et al. 2011). Prior work introduces a record and replay approach that designs (1) an *optimistic recorder* on top of optimistic tracking and (2) an *optimistic replayer* for the recorder (Bond et al. 2013, 2015). (The optimistic replayer is "optimistic," because it replays dependencies recorded by the optimistic recorder. It does *not* use optimistic tracking.) The optimistic recorder identifies and records happens-before edges at transitions among $\mathrm{WrEx}^{\mathrm{Opt}}$, $\mathrm{RdEx}^{\mathrm{Opt}}$, and $\mathrm{RdSh}^{\mathrm{Opt}}$ states. It records each happens-before edge by recording the edge's *source* and *sink* in per-thread logs. In another execution, the optimistic replayer replays each happens-before edge by making the sink wait for its corresponding source to be reached.

3.2.2 Hybrid Dependence Recorder and Replayer. We design a *hybrid recorder* based on HT 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 $\mathrm{RdEx}_{\mathrm{T}}^{\mathrm{Pess}} \rightarrow \mathrm{RdSh}_{\mathrm{c}}^{\mathrm{RLock(2)}}$ in the same way that it records $\mathrm{RdEx}_{\mathrm{T}}^{\mathrm{Opt}} \rightarrow \mathrm{RdSh}_{\mathrm{c}}^{\mathrm{Opt}}$.

Pessimistic conflicting transitions. The key challenge is pessimistic transitions that involve conflicting states (i.e., a transition from $\mathrm{WrEx}_{\mathrm{T1}}^{\mathrm{Pess}}$ to $\mathrm{WrEx}_{\mathrm{T2}}^{\mathrm{WLock}}$ or $\mathrm{RdEx}_{\mathrm{T2}}^{\mathrm{RLock}}$ or from $\mathrm{RdEx}_{\mathrm{T1}}^{\mathrm{Pess}}$ or $\mathrm{RdSh}_{\mathrm{c}}^{\mathrm{Pess}}$ to $\mathrm{WrEx}_{\mathrm{T2}}^{\mathrm{WLock}}$). Figure 5(a) shows an example that illustrates why these transitions are problematic. For this example, suppose pessimistic transitions *do not* defer unlocking. Thread T1 immediately unlocks an object o to $\mathrm{WrEx}_{\mathrm{T1}}^{\mathrm{Pess}}$ 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, thus recording cyclic dependencies that cannot be replayed successfully.

In contrast, an *optimistic* conflicting transition triggers coordination, as shown in Figure 5(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 3.1.1), 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 3.1.1). Deferred unlocking of pessimistic states effectively limits the potential sources of happens-before edges to PSROs and responding safe points.

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 3(a) from Section 3.1.1 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 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 3(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.

3.3 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 dependencies.

3.3.1 Optimistic RS Enforcer. Modern language memory models make strong guarantees for DRF programs but provide virtually no guarantees for programs with data races (Boehm and Adve 2008; Manson et al. 2005; Adve and Hill 1990; Adve and Boehm 2010; Boehm 2012). Prior work enforces memory models that provide RS even for programs with data races (Ouyang et al. 2013; Sengupta et al. 2015; Hammond et al. 2004). 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 (Sengupta et al. 2015).

Prior work, which we call the *optimistic enforcer*, enforces SBRS using optimistic tracking at each object access (Sengupta et al. 2015). The optimistic enforcer provides region serializability via

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

Fig. 6. The challenge of building an RS enforcer using HT.

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 10 and 19 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 execute either idempotently or speculatively and can thus restart safely after responding to a coordination request, as prior work describes in detail (Sengupta et al. 2015).

3.3.2 Hybrid RS Enforcer. To understand the challenges of using HT 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 6. The compiler cannot predict whether the accesses to objects o and p will use pessimistic versus optimistic tracking, the end of the region needs conditional code that checks which pessimistic states to unlock, if any. Assuming most accesses will be optimistic, most regions would need to unlock *no* pessimistic states. Since 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 3.1.1).

Using deferred unlocking. Our *hybrid enforcer* relies on deferred unlocking to address these challenges. HT defers unlocking of pessimistic states until PSROs. PSROs are relatively infrequent compared to region boundaries, so it is relatively 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 a thread that initiates coordination might respond to other threads' coordination requests, a thread restarts a region that performs coordination, just as it does for *optimistic* conflicting transitions.

3.4 Adaptive Policy

This section addresses how to choose between pessimistic and optimistic states at runtime. 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.

3.4.1 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,[4] optimistic conflicting,[5] 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. To minimize runtime, 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 Equation (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.

Substituting Equation (1) into Equation (2) and transforming it yields:

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

where K_{confl} is a runtime constant:

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

Thus, according to Equation (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.

3.4.2 Profile-Guided Adaptive Policy. Using the cost–benefit model to change each object's state to optimistic or pessimistic at runtime 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.

Efficient profiling. Counting optimistic same-state transitions would be expensive, because they are common (by design). The adaptive policy thus profiles only conflicting transitions for optimistic objects,[6] and it counts all pessimistic transitions, since they are relatively infrequent (by design). The resulting policy readily transfers potentially high-conflict objects to pessimistic states—at which point more-invasive profiling categorizes every pessimistic transition in order to determine whether an object should stay in pessimistic states or change back to optimistic states.

[4]The model computes $T_{nonConfl}$ 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.
[5] T_{confl} is the time for a conflicting transition using *explicit* coordination.
[6]The policy counts only transitions that use explicit coordination, since implicit coordination is roughly as expensive as a pessimistic transition.

For each object o, the profiling counts the number of optimistic conflicting transitions o.numConflicts. If an object experiences "enough" conflicting transitions, that is, if

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

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 Eqution (3):

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

The parameter *Inertia* avoids prematurely changing back to optimistic states until 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 Equation (4). Then profile-guided decisions based on Equation (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 the $\text{WrEx}_T^{\text{Opt}}$ 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.

Per-object profiling. 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 6.2.3). Thus, our implementation uses only per-object profiling.

4 RELAXED TRACKING AND RUNTIME SUPPORT

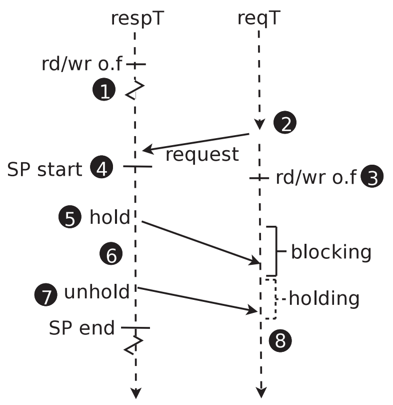
Pessimistic and optimistic tracking (Section 2), as well as hybrid tracking (Section 3), track each cross-thread dependence soundly (i.e., do not miss any dependencies) and preserve instrumentation–access atomicity. We thus classify them as *strict* dependence tracking approaches.

In contrast, this section introduces a novel approach called RT that relaxes the instrumentation–access atomicity guarantee of optimistic tracking to reduce its coordination latency. RT allows threads to continue executing program code without receiving coordination response(s) for a state change. The challenge in making RT work lies in preserving both program semantics and runtime-support-specific guarantees. Section 4.1 presents RT, and Sections 4.2 and 4.3 present runtime support based on RT.

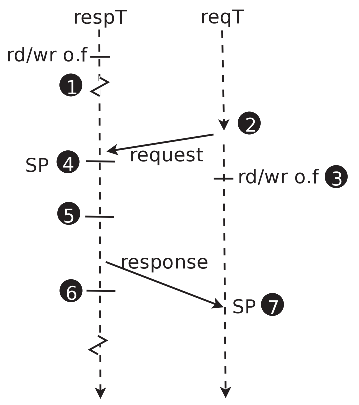
4.1 Relaxed Tracking

RT consists of two components: a *relaxed coordination protocol* (Section 4.1.1) and support for performing *relaxed accesses* that overlap with coordination (Section 4.1.2).

4.1.1 The Relaxed Coordination Protocol. RT modifies the way in which optimistic tracking handles an explicit coordination request. In RT, a requesting thread does *not* wait for responses after sending explicit requests. Thus, a requesting thread receives responses at some later point in its execution, and a requesting thread may have outstanding requests for multiple objects



(a) **Implicit response:** (1) respT accessed o at some prior time. (2) reqT wants to access o. It changes o's state to $\mathsf{RdEx}^{\mathsf{Int}}_{\mathsf{reqT}}$ or $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{reqT}}$. (3) reqT proceeds without waiting to receive respT's response. (4) respT reaches a safe point (SP) and (5) sees reqT in a blocking state, so respT puts reqT in a "blocked and held" state. (6) respT changes o's state to $\mathsf{WrEx}_{\mathsf{reqT}}$ or $\mathsf{RdEx}_{\mathsf{reqT}}$. (7) respT removes the hold on reqT, then leaves the safe point. (8) reqT finishes blocking, then waits until all holds have been removed.



(b) **Explicit response:** (1) respT accessed o at some prior time. (2) reqT wants to access o. It changes o's state to $\mathsf{RdEx}^{\mathsf{Int}}_{\mathsf{reqT}}$ or $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{reqT}}$. (3) reqT proceeds without waiting to receive respT's response. (4) respT reaches a safe point (SP), (5) sends an explicit response, and (6) leaves the safe point. (7) reqT reaches a safe point and sees the response. reqT changes o's state to $\mathsf{WrEx}_{\mathsf{reqT}}$ or $\mathsf{RdEx}_{\mathsf{reqT}}$.

Fig. 7. The *relaxed* coordination protocol (for explicit requests only).

simultaneously. To support this functionality, the relaxed coordination protocol differs from optimistic tracking's strict coordination protocol (Section 2.2) in the following ways:

—A responding thread can *respond* either implicitly or explicitly, depending on whether the *requesting* thread is blocking or actively executing program code.
—To support explicit responses, relaxed coordination extends strict coordination's request queue to a *request-and-response queue* that holds both requests and responses. At safe points, threads can receive not only requests but also responses.

Figure 7 shows how the relaxed coordination protocol works. The requesting thread reqT sends an explicit request to the responding thread respT and continues execution. When respT reaches a safe point, it responds to reqT either explicitly or implicitly. If reqT is blocked, then respT responds implicitly, as shown in Figure 7(a), by first putting reqT into a "blocked and held" state (so reqT does not leave the blocking state unless it is "unheld") and then changing the object's state. Finally, the responding thread removes its hold on the requesting thread. Otherwise (reqT is not blocked), respT responds explicitly, as Figure 7(b) shows, by adding a response to reqT's queue. Once reqT reaches a safe point, it changes the object's state. Although Figure 7 shows a single requesting thread sending requests to respT, multiple requesting threads can send requests to respT before respT reaches a safe point. When respT reaches a safe point, it responds to each queued request in turn.

For a conflicting transition from $\mathsf{WrEx}_{\mathsf{respT}}$ or $\mathsf{RdEx}_{\mathsf{respT}}$ to $\mathsf{WrEx}_{\mathsf{reqT}}$ or $\mathsf{RdEx}_{\mathsf{reqT}}$, reqT receives just one response. In contrast, for a transition from RdSh_{c} to $\mathsf{WrEx}_{\mathsf{reqT}}$, reqT may need to wait for multiple responses. The protocol maintains a counter of unreceived responses for each object in this situation, which responding and requesting threads decrement as they respond implicitly and receive explicit responses, respectively. In all cases, the requesting thread maintains the object in

```
 1 if (o.state != WrEx$_T$) {
 2   writeSlowPath(o, &o.f, newValue);
 3 } else {
 4   o.f = newValue; // original program store
 5 }

 6 writeSlowPath(o, addr, newValue) {
 7   state = o.state;
 8   // Handle non−conflicting state transitions :
 9   if (state == ... ) {
10     ... ; ∗addr = newValue; return;
11   }
12   if (state == RdEx$_T^{Int}$) {
13     /∗ upgrading trans. to WrEx$_T^{Int}$ ∗/
14   }
15   boolean relaxed = true;
16   while (state != WrEx$_T^{Int}$) {
17     if (state != WrEx$_*^{Int}$ &&
18         CAS(&o.state, state, WrEx$_T^{Int}$)) {
19       relaxed = !sendRequest(getOwner(state));
20       break;
21     }
22     state = o.state; // re−read state
23   }
24   if (relaxed) {
25     // defer the store
26     storeBufferSet(addr, newValue);
27   } else {
28     o.state = WrEx$_T$;
29     ∗addr = newValue;
30   }
31 }
```

Fig. 8. The fast and slow paths of RT's instrumentation at stores.

```
 1 if (o.state != WrEx$_T$ &&
 2     o.state != RdEx$_T$ &&
 3     (o.state != RdSh$_c$ || T.rdShCount < c))
 4   ... = readSlowPath(o, &o.f); {
 5 } else {
 6   ... = o.f; // original program load
 7 }

 8 readSlowPath(o, addr) {
 9   state = o.state;
10   // Handle non−conflicting state transitions :
11   if (state == ... ) { ... ; return ∗addr; }
12   if (state == WrEx$_T^{Int}$ &&
13       storeBufferHas(addr))
14     return storeBufferGet(addr);
15   boolean relaxed = true;
16   while (state != WrEx$_*^{Int}$ && state != RdEx$_*^{Int}$) {
17     // Coordination for conflicting transition :
18     if (CAS(&o.state, state, RdEx$_T^{Int}$)) {
19       relaxed = !sendRequest(getOwner(state));
20       break;
21     }
22     state = o.state; // re-read state
23   }
24   value = ∗addr;
25   if (relaxed)
26     logLoadedValue(addr, value);
27   else
28     o.state = RdEx$_T$;
29   return value;
30 }
```

Fig. 9. The fast and slow paths of RT's instrumentation at loads.

an intermediate state, WrEx$^{\text{Int}}_{\text{reqT}}$ or RdEx$^{\text{Int}}_{\text{reqT}}$, until it has received all response(s). In the meantime, no other thread can change the object's state.

Figures 8 and 9 show the pseudocode for RT's load and store instrumentation. RT uses the same fast path as optimistic tracking (Section 2.2), except that it skips the original program access if it takes the slow path, delegating the access to the slow path instead. For both loads and stores in the slow path, RT first changes the object's state to an intermediate state, WrEx$^{\text{Int}}_{\text{T}}$ (line 18 in Figure 8) or RdEx$^{\text{Int}}_{\text{T}}$ (line 19 in Figure 9), and initiates coordination by sending a request to the responding thread (line 19 in Figure 8 and line 19 in Figure 9). After sending the coordination request, T continues execution immediately.

Since T does not wait for responses, it instead receives responses at safe points (not shown). A responding thread *responds* either implicitly or explicitly, depending on whether or not the requesting thread is at a blocking safe point. Before T receives a response, the conflicting object o stays in the WrEx$^{\text{Int}}_{\text{T}}$ or RdEx$^{\text{Int}}_{\text{T}}$ state, since both T and other threads might perform accesses to it. If the access is a store, and o is in RdEx$^{\text{Int}}_{\text{T}}$ state, then RT upgrades the state to WrEx$^{\text{Int}}_{\text{T}}$ (line 13 in Figure 8), in order to track relaxed stores, introduced shortly. If the access is a store and o's state is WrEx$^{\text{Int}}_{*}$ but not WrEx$^{\text{Int}}_{\text{T}}$ (i.e., another thread is performing relaxed stores to o), then the access needs to wait until the state has changed to a non-intermediate state. If the access is a load, as long as o is in WrEx$^{\text{Int}}_{*}$ or RdEx$^{\text{Int}}_{*}$ (i.e., an intermediate state for any thread), then T can avoid performing coordination and proceed to perform the load.

We note that RT's relaxed coordination protocol differs from the strict coordination protocol for *explicit* requests only. In RT, when coordination uses an *implicit* request, it follows the same steps as strict coordination.

Interestingly, the relaxed coordination protocol handles requests and responses in a largely symmetric way. Requests and responses each involve sending a message to another thread, either implicitly if the receiving thread is at a blocking safe point or else explicitly via a queue that the receiving thread processes at its next safe point.

4.1.2 Handling Relaxed Accesses. A thread T performs *relaxed accesses*[7] to objects whose states are not (yet) in the needed state. RT defers a *relaxed store* until it receives coordination response(s) for the object's state. As we explain, relaxed stores still conform to the language memory model as long as they are not deferred past synchronization release operations. RT performs a *relaxed load* by loading from an object before receiving coordination response(s) for the object's state. Relaxed loads do not affect program correctness, but they can affect runtime support's guarantees.

Relaxed stores. A thread T performs a relaxed store by *deferring* the store, buffering the location (address) and new value in T's *store buffer* (line 26 in Figure 8). The intuition behind deferring stores is that another thread may be simultaneously (racily) accessing the same location, so allowing the store to be performed could cause a cross-thread dependence to be missed. Once T gets exclusive ownership of the conflicting object o (by changing o's state to $\mathsf{WrEx_T}$), it commits all relaxed stores to o using the store buffer. For simplicity, our current design allows relaxed stores by T only to objects in $\mathsf{WrEx_T^{Int}}$ state. (We have found that supporting relaxed stores to other intermediate states provides little benefit.)

Deferring program stores changes program behavior, since other threads can read out-of-date values from the affected memory locations. However, language memory models, including for Java and C++, allow substantial reordering of operations, except across synchronization operations (Adve and Boehm 2010; Boehm and Adve 2008; Manson et al. 2005; Adve and Hill 1990), thus permitting significant deferring of stores. To conform to the memory model and preserve program semantics, the key constraint is that stores *cannot* be deferred past PSROs.

The relaxed coordination protocol completes in bounded time after changing an object's state to $\mathsf{WrEx_{reqT}^{Int}}$, since threads execute safe points within a bounded amount of time. After the protocol completes and the requesting thread changes the object's state to $\mathsf{WrEx_{reqT}}$, the requesting thread removes the object from its store buffer. Relaxed stores thus become visible to other threads in bounded time.

Relaxed loads. At a relaxed load by T to an object o, T first checks whether the same location has already been buffered in T's store buffer (line 13 in Figure 9). (T only needs to check its store buffer if o's state is $\mathsf{WrEx_T^{Int}}$.) If so, then T uses the store buffer's value (line 14 in Figure 9) instead of loading from memory. Otherwise, T performs the load directly from memory (line 24 in Figure 9). A relaxed load thus does *not* affect program semantics: The execution still conforms to the memory model (performing the load would be permitted in the original program). However, another thread might be simultaneously (racily) writing to the same memory location, compromising the ability of runtime support to capture the write–read or read–write dependence soundly.

RT thus handles each relaxed load by *logging the loaded value* in a *runtime-support-specific way* (line 26 in Figure 9). The intuition is that logging the value enables runtime support to handle all values resulting from potentially untracked cross-thread dependencies. For example, our relaxed

[7]This article's *relaxed accesses* should not be confused with memory_order_relaxed operations on atomic variables in C/C++ (Boehm and Adve 2008).

dependence recorder logs the value in order to assist replay (Section 4.2), and our relaxed STM logs the value in order to validate it later (Section 4.3).

4.1.3 Optimizations at PSROs. As presented so far, a thread must wait at each PSRO for every outstanding relaxed store (i.e., every entry in its store buffer). This restriction limits RT's ability to overlap coordination with program execution; we have found that threads routinely end a critical section (by releasing a lock) shortly after performing a store to a shared variable. Here we present two optimizations for avoiding waiting at PSROs. As our evaluation shows, these optimizations have performance benefits but also drawbacks that lead to mixed performance relative to the base RT design described so far.

Defer release operations. Instead of waiting at a release operation for outstanding relaxed stores, a thread can *defer the release operation.* Our design currently supports deferring *lock* release operations. To defer a lock release, a thread continues to hold the lock past the release operation; the thread records the lock into a per-thread *lock buffer.* It releases the lock only when all responses have been received for relaxed stores that executed before the lock release (according to bookkeeping) or right before an acquire that tries to hold a different lock to avoid change lock ordering. This technique should not to be confused with deferred unlocking in HT (Section 3.1.1), which defers changing objects' states instead of releasing program locks.

This behavior naturally preserves program semantics, because a thread continues to hold a lock while it waits for responses for relaxed stores, effectively expanding the lock's critical section—making other threads wait and thus increasing lock contention. Effectively enlarging critical sections can serendipitously avoid some erroneous behaviors, which may be desirable or undesirable, depending on the goals and setting.

Avoid stalling at release. An alternate approach is to permit a thread T to continue execution at a release operation—as long as no other thread may access the object(s) that are the targets of relaxed stores. A straightforward way to provide this restriction is to disallow all accesses by other threads to objects in $\mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Int}}$ state. (Note that, in contrast, the base RT design allows loads, but not stores, to an object in any intermediate state.) This optimization allows threads to continue without waiting at release operations, but it incurs other costs, because a thread T2 must wait to access an object in the $\mathsf{WrEx}_{\mathsf{T1}}^{\mathsf{Int}}$ state.

4.2 Relaxed Dependence Recorder

This section introduces a *relaxed dependence recorder* for multithreaded record and replay. Our relaxed recorder extends the optimistic recorder from prior work (Section 3.2.1) by using relaxed, instead of strict, dependence tracking. The relaxed recorder allows relaxed loads and stores to objects that are not yet "owned" by the current thread. Given this behavior, how is it possible to record dependencies accurately and thus guarantee deterministic replay?

We refer back to Figure 7 in Section 4.1 for examples of happens-before edges recorded by the relaxed recorder. For an implicit response, at point #6, respT records the source of the edge. At point #8, reqT records the edge's sink. For an explicit response, respT records the source at point #5, and reqT records the sink at point #7. Note that if the replayed execution replays these same edges, it will not necessarily reproduce the same behavior, because the relaxed accesses at point #3 (and other relaxed accesses potentially overlapping with coordination) are not ordered by the edge. The key to addressing this problem is to record enough information about loads and stores that are *not* well-ordered by happens-before edges, such that they can be replayed faithfully.

Handling stores. To handle relaxed stores to objects in the $\mathsf{WrEx}_{\mathsf{reqT}}^{\mathsf{Int}}$ state, the relaxed recorder uses the following strategy: reqT records an event for each relaxed store to indicate that the store should also be deferred *during replay*. When stores are performed from the store buffer at a safe point, reqT records an event indicating that relaxed stores should be performed at that safe point. By referring to indices of entries in the store buffer, the recorded event unambiguously indicates *which* stores should be performed from the store buffer at each safe point during replay.

Handling loads. When a requesting thread reqT loads a value from an object that is in an intermediate state, the responding thread may be simultaneously writing the object. Thus, it does *not* seem possible to record a happens-before edge that will yield the same value for the load. Instead, reqT *records the value* returned by the load (at point #3 in both Figure 7(a) and Figure 7(b)). A replayed execution can reuse this value to ensure determinism.

During the recorded execution, subsequent loads to the same memory location record the (possibly updated) loaded value. Whenever reqT handles a load by getting the value from its store buffer (Section 4.1.2), it still records the value in its log, so a replayed execution can load the correct value without needing to know which loads should read from the store buffer.

We have not built a replayer for the relaxed recorder due to an engineering challenge in handling relaxed loads of *reference values* (i.e., references to objects). The base record and replay system provides *application-level determinism*, which does *not* guarantee each object is located at the same address in the replayed execution (Bond et al. 2015). Thus, a logged reference value from the recorded execution is unlikely to point to the same object in the replayed execution. We could address this challenge by instead recording a unique identifier for the referenced object; the record and replay system already provides unique identifiers for objects, which consist of the allocating thread and the value of a per-thread allocation counter to provide deterministic hash codes (Bond et al. 2015). The replayed execution could use the logged identifier to get a reference to the correct object in the replayed execution. In the common case, the replayed execution can check whether performing the load normally would return a reference to the correct object (the object with the logged identifier). If not, then the replayed execution would need to use a mechanism to look up objects based on their identifiers.

4.3 Software Transactional Memory

This section describes how we extend an existing STM system to use RT. In essence, our *relaxed STM* combines lazy and eager concurrency control in a novel way: It uses eager mechanisms for most accesses and lazy mechanisms for accesses that would otherwise incur latency.

4.3.1 Optimistic STM. Prior work introduces an STM that uses biased reader–writer locks that employ optimistic tracking (Zhang et al. 2015). We call this STM the *optimistic STM*. The optimistic STM employs biased reader–writer locks to provide eager concurrency control: It detects and resolves conflicts before performing each memory access. Conflict detection and resolution piggyback on coordination.

Here we focus on how the STM piggybacks on coordination that uses an *explicit* request. In that case, the *responding* thread detects and resolves conflicts between the responding thread's transaction and the requesting thread's transaction or non-transactional access.

4.3.2 Relaxed STM. Extending the optimistic STM to use RT presents challenges. Unless handled properly, the STM could be unable to detect and resolve transactional conflicts for relaxed loads and stores. Figure 10 shows an example of a problematic execution. Thread T2 performs two relaxed loads from o.f in a transaction, since o's state is $\mathsf{WrEx}_{\mathsf{T1}}$. T1 performs conflict detection when it responds to T2, but by then T1 has started another transaction that has not accessed o, so T1 accurately reports no transactional conflict. However, the result is unserializable, because

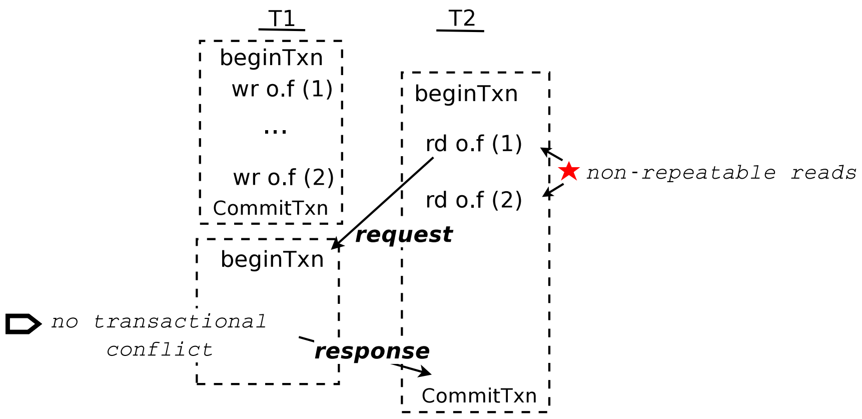


Fig. 10. Allowing unhandled relaxed accesses in transactions would lead to serializability violations. The values in parentheses after each executed store and load are the values written and read, respectively.

T2's loads see different values. Another problematic issue (not shown) is that performing relaxed transactional stores directly could lead to unserializable results due to another thread loading the value simultaneously.

Our *relaxed STM* addresses these issues as follows. In the relaxed STM, each relaxed, transactional load logs its loaded value and *validates* the value later. (In Figure 10, T2's transaction would fail read validation at commit time, and abort.) The relaxed STM buffers relaxed stores until coordination responses have been received (i.e., RT's default behavior), providing opacity of transactional updates. Before a transaction commits, it waits for coordination responses so it can validate all relaxed loads and perform any outstanding relaxed stores.

Relaxed loads. A requesting thread reqT performs a relaxed load when it reads from an object o in the $\text{WrEx}^{\text{Int}}_{\text{reqT}}$ or $\text{RdEx}^{\text{Int}}_{\text{reqT}}$ state. reqT first checks its store buffer for the value (only if the object's state is $\text{WrEx}^{\text{Int}}_{\text{reqT}}$). If not found, then reqT performs the load—but responding thread(s) may be simultaneously writing to o, potentially violating serializability. reqT thus logs the loaded location and value in a *read validation log*.

When reqT receives the response for o, it validates all entries in the read validation log against o's *current* value(s), as the following pseudocode shows:

```
foreach (addr, value) in readValidationLog
   if (*addr != value) abortTxn();
```

For every field or array element of o in the read validation log, the current value must match the log's value. This logic makes sense as follows. The responding thread responded at some safe point where it performed conflict detection (and potentially conflict resolution). Validating the loaded value ensures that the values that were read previously for o are the same as if the values had all been read at the responding thread's responding safe point. Even if another thread changed a location to a different value and back to the original value (i.e., the "ABA problem"), the transaction is still serializable as long as its read(s) are consistent with the commit-time value, as for value validation in NOrec (Dalessandro et al. 2010). If validation fails, then reqT must abort its current transaction. (If respT responds implicitly, then it performs the above work on behalf of reqT.)

Relaxed stores. A requesting thread reqT defers a relaxed store by buffering its location (address) and value in the store buffer, which is analogous to the *redo log* used by STMs that use lazy versioning (Harris et al. 2010). After reqT receives all responses for o, it performs the store(s) for o from the store buffer—and also logs the store(s) in the *undo* log. (If respT responds implicitly, then it performs all of these actions on behalf of reqT.)

Commit and abort. Before a transaction commits or aborts, it waits for all outstanding responses, in order to validate loads and perform outstanding relaxed stores. Unlike our general RT design, for the relaxed STM, RT does *not* allow loads by T to objects in intermediate states *other than* $\mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Int}}$ and $\mathsf{RdEx}_{\mathsf{T}}^{\mathsf{Int}}$ (loads must wait for a state change); supporting loads from other states would require a mechanism for eventually changing the state to $\mathsf{WrEx}_{\mathsf{T}}$, $\mathsf{RdEx}_{\mathsf{T}}$, or RdSh to validate reads before commit.

Guaranteeing progress. The *optimistic* STM guarantees progress by detecting all conflicts eagerly and then aborting the younger transaction (Spear et al. 2009; Zhang et al. 2015). However, the *relaxed* STM cannot guarantee progress, since any transaction that fails read validation must abort. Other mixed-mode STMs have similarly lacked progress guarantees (Harris et al. 2006; Saha et al. 2006) (fully lazy STMs such as NOrec (Dalessandro et al. 2010) can ensure progress easily). Standard techniques such as exponential backoff can help to alleviate livelock. For the relaxed STM, there exists a simple (unimplemented) solution (which resembles solutions from prior work (Ni et al. 2008; Sonmez et al. 2009): If a transaction repeatedly fails read validation, then it falls back to use *strict* dependence tracking, guaranteeing it will commit (at least once it becomes oldest).

Correctness. At a high level, the relaxed STM provides serializability by guaranteeing that all of a committing transaction's operations appear as though they happened instantaneously at commit time. For conflicting accesses handled by optimistic tracking, eager conflict detection and resolution guarantee that conflicting accesses between the committing transaction's accesses and commit time will be detected and resolved. (The relaxed STM uses the same mechanism for relaxed stores but defers making the store visible until relaxed coordination has finished.) For relaxed loads, commit-time value validation ensures that each value from a relaxed load is consistent with the commit-time value of the memory location.

Semantics. Lazy read validation can lead to so-called *zombie* transactions whose behavior is impossible in any serializable execution (Harris et al. 2010). In managed languages such as Java, zombies are not a serious problem, because memory and type safety are preserved (Menon et al. 2008; Dalessandro and Scott 2012). Targeting a native language such as C++ would require additional support to provide *sandboxing* of zombie transactions (Dalessandro and Scott 2012). Another issue is that zombie transactions can get stuck in infinite loops that are impossible in any serializable execution. The relaxed STM cannot experience this issue, because it validates relaxed loads within a bounded amount of time.

Comparison with prior work. Relaxed loads and stores in the relaxed STM are, in essence, the same as lazy mechanisms employed by some STMs (e.g. Spear et al. (2009), Dalessandro et al. (2010), Dragojević et al. (2009), Zhang et al. (2015), Olszewski et al. (2007), Saha et al. (2006), and Harris et al. (2006)). Some STMs have even combined lazy and eager mechanisms, by using eager concurrency control for *writes* and lazy validation for *reads* (Saha et al. 2006; Harris et al. 2006). However, the relaxed STM combines eager and lazy concurrency control in a novel way, using eager and lazy concurrency control for non-conflicting and conflicting accesses, respectively.

5 IMPLEMENTATION

We have implemented HT and RT, and the hybrid recorder and replayer, hybrid RS enforcer, relaxed recorder, and relaxed STM in Jikes RVM 3.1.3 (Alpern et al. 2000, 2005), a Java virtual machine that performs competitively with commercial JVMs (Biswas et al. 2015). Since HT and RT are in separate branches of our code base, we refer to HT and hybrid runtime support as one implementation and to RT and relaxed runtime support as the other implementation.

We have made our implementations publicly available on the Jikes RVM Research Archive. Our implementations build on publicly available implementation of pessimistic and optimistic tracking (Bond et al. 2013), the optimistic recorder and replayer (Bond et al. 2015), the optimistic RS enforcer (Sengupta et al. 2015), and the optimistic STM (Zhang et al. 2015). Our implementations reuse features of existing implementations as much as possible.

By targeting a managed language, our implementations 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.

The implementations modify Jikes RVM's dynamic just-in-time compilers to insert instrumentation before every heap memory access, PSRO, and safe point in the application and Java libraries.

Since x86-64 support in Jikes RVM is still a work in progress, our implementation targets the IA-32 platform. The HT implementation adds two 32-bit words to each (scalar and array) object and to each 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. The RT implementation uses the same state model as optimistic tracking, using one 32-bit word for each object and for each static field.

HT maintains a per-thread lock buffer and read set to support deferred unlocking of pessimistic states (Section 3.1.2). The HT implementation uses a sequential store buffer for the lock buffer and a lightweight hash table for the read set to support efficient lookups.

RT maintains each thread's store buffer as a sequential store buffer. Lookup time (by relaxed loads) is linear in the size of the buffer; however, the buffer size is equal to the number of relaxed stores in a synchronization-free region, which is small in practice. The relaxed STM logs relaxed loads (addresses and values) in per-thread sequential store buffers.

Although HT and RT are potentially complementary, we have not endeavored to combine them together, due to the complexity and challenges of doing so. For example, combining them would require resolving a complex mismatch between pessimistic and relaxed tracking—more complex than the mismatch between pessimistic and (strict) optimistic tracking. Furthermore, designing correct runtime support would be more complicated than for HT or RT alone.

Extraneous contention in HT. Due to limited bit patterns available in a metadata word, the prototype HT 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, that is, 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 an object in $\mathsf{WrEx}_\mathsf{T1}^\mathsf{Pess}$ state 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 dependencies soundly. This *unsound* configuration provided no performance benefit, indicating that the *default* configuration is not encountering significant spurious contention in our experiments.

6 EVALUATION

This section evaluates the runtime characteristics and performance of HT and RT, compared with pessimistic and optimistic tracking alone. It also compares the performance of the runtime support based on HT and RT, with the corresponding optimistic versions.

6.1 Methodology

Benchmarks. The experiments execute the following benchmarks:

- *Benchmarked versions of large, real programs:* the DaCapo benchmarks, versions 2006-10-MR2 and 9.12-bach (2009) (Blackburn et al. 2006), excluding single-threaded programs and programs that Jikes RVM cannot execute
- *Business logic benchmarks:* fixed-workload versions of SPECjbb2000 and SPECjbb2005[8]
- *Transactional benchmarks:* the STAMP benchmarks (Cao Minh et al. 2008), ported to Java, providing six working programs (Demsky and Dash 2010; Korland et al. 2010; Zhang et al. 2015).

The original RT article includes results for additional benchmarks (Zhang et al. 2016), which this article omits.

Experimental setup. For each implementation, we build a high-performance configuration (FastAdaptiveGenImmix) of Jikes RVM. Each performance result is the median of at least 20 trials. We also show the mean, as the center of 95% confidence intervals. Each reported statistic is the mean from five statistics-gathering runs.

Platform. Experiments execute on a machine with 4 Intel Xeon E5-4620 8-core processors with hyperthreading disabled (32 cores total) running Linux 2.6.32.

6.2 Evaluating Hybrid Tracking

HT's experiments use the following adaptive policy parameter values: $Cutoff_{confl} = 4$, $K_{confl} = 200$, $Inertia = 100$. As explained earlier in Section 3.4, we use a low value for $Cutoff_{confl}$. We found that larger values of $Cutoff_{confl}$ hurt the performance of avrora9 but otherwise have little impact (results not shown). We also found that performance is not very sensitive to the other parameters; various values for K_{confl} (20–1,600) and *Inertia* (20–1,600) are effective (results not shown).

6.2.1 Runtime Characteristics. Table 2 counts state transitions under HT. 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. HT 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

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

Table 2. State Transitions for HT and Optimistic Tracking

	Optimistic transitions				**Pessimistic transitions**			**Opt. to**	**Pess.**
	Same state		Conflicting		Uncont.	%Reentr.	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}

Note: For comparison, state transitions for optimistic tracking alone are shown in parentheses.

a performance loss, since a conflicting transition costs two to three 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 HT. We note that deferred unlocking enables a significant fraction of uncontended (*Uncont.*) accesses to be reentrant (*Reentr.*) 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 deferred unlocking of pessimistic states is generally successful. However, for avrora9 and pjbb2005, contended transitions are of the same order as optimistic conflicting transitions, so HT 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.

6.2.2 Performance of Tracking Alone. Figure 11 compares the performance of HT with pessimistic and optimistic tracking alone (without runtime support on top of dependence tracking). Each bar shows the runtime 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.

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

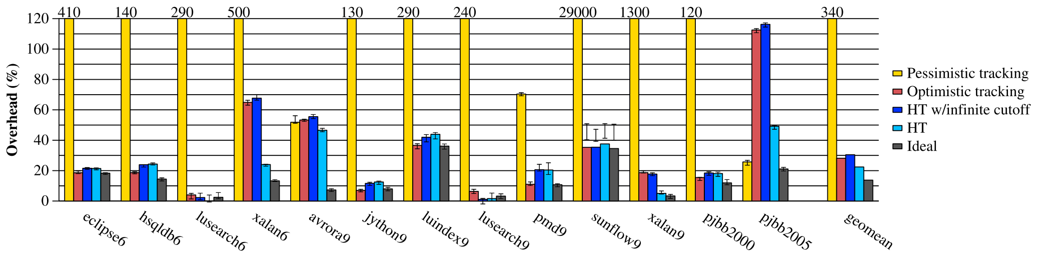


Fig. 11. Runtime overhead of pessimistic and optimistic tracking compared with HT. Each bar is the median of 20 trials. The intervals are 95% confidence intervals centered at the mean. Overheads exceeding 120% are labeled using two significant figures.

HT uses the default values of *Cutoff*$_{confl}$ and other parameters. HT 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). HT reduces overhead by 63% (65% → 24%) for xalan6, by 74% (19% → 5%) for xalan9, and by 45% (110% → 49%) for pjbb2005. Despite reducing conflicting transitions significantly for hsqldb6 (Table 2), HT 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 HT might be able to provide.

HT adds 22% average overhead, 21% less than optimistic tracking's 28% overhead. HT incurs 27% less overhead than *HT 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, HT provides better performance for high-conflict programs. On average, HT adds lower overhead than both pessimistic and optimistic tracking alone.

Many of the programs we evaluate perform relatively little shared-memory communication (Kalibera et al. 2012). These programs may or may 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 HT to improve. Nevertheless, only HT can scale to diverse communication patterns: It helps cases for which optimistic tracking performs poorly, without harming cases for which optimistic tracking performs well.

6.2.3 Limit Study for the Adaptive Policy's Per-Object Policy. To evaluate whether per-object profiling identifies most optimistic conflicting transitions in advance, we perform a limit study on optimistic tracking alone. Figure 12 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 fraction of overall program accesses. For high-conflict programs, most conflicting transitions are to objects that have triggered many conflicting transitions

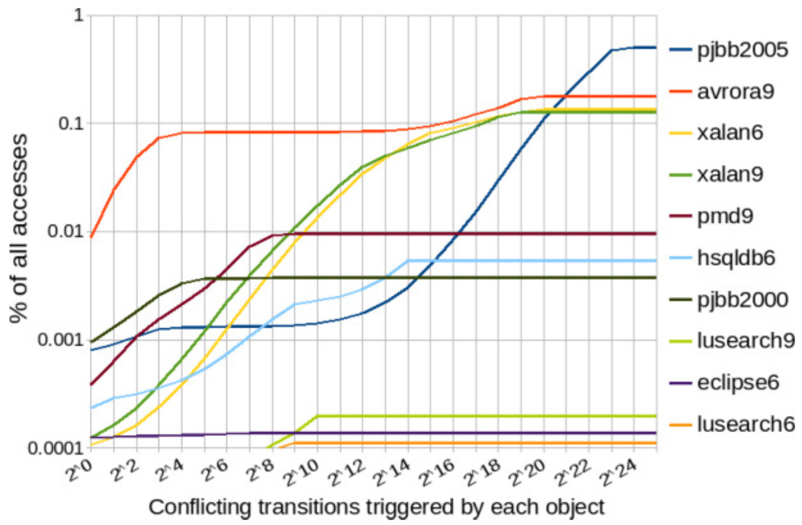
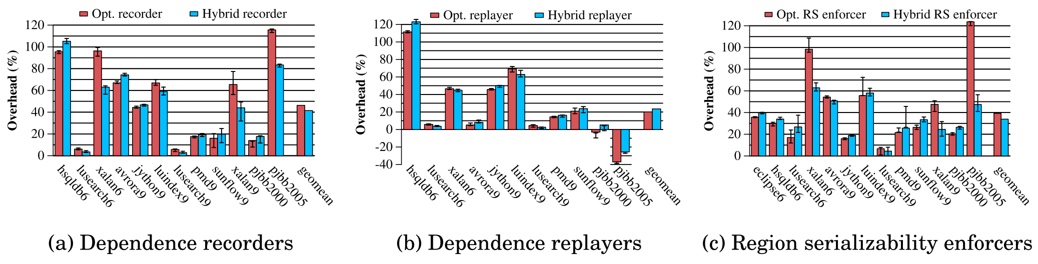


Fig. 12. Cumulative distribution of conflicting transitions (explicit coordination only) triggered per object for optimistic tracking. Both axes use a logarithmic scale. The legend sorts programs by their maximum y-axis value. Three programs have a conflict rate <0.0001% and are excluded.



(a) Dependence recorders (b) Dependence replayers (c) Region serializability enforcers

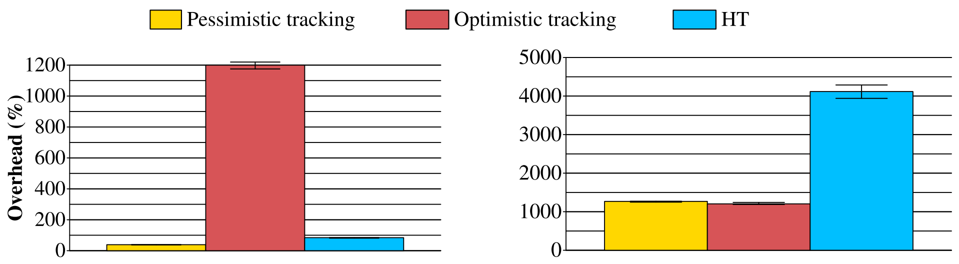
Fig. 13. runtime overhead of optimistic and hybrid runtime support.

(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.

6.2.4 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 13(a) shows the performance of the optimistic and hybrid dependence recorders. HT 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 dependencies as the optimistic recorder does. This fact explains why the hybrid recorder’s improvement over the optimistic recorder is smaller than for HT over optimistic tracking alone.

Figure 13(b) shows the performance of optimistic and hybrid replayers. The optimistic *replayer* is not fully robust: It successfully replays 11 of 13 programs (failing on eclipse6 and xalan9) (Bond



(a) syncInc

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

(b) racyInc

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

Fig. 14. Runtime overhead of tracking alone on microbenchmarks.

et al. 2015). The optimistic replayer adds 20% overhead on average—lower than the optimistic recorder—because it is cheaper to replay known dependencies than record unknown dependencies. 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 dependencies, so it can outperform baseline execution for programs dominated by coarse-grained, overly conservative synchronization.

The 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 HT cannot reduce the number of replayed cross-thread dependencies. Overall, HT improves 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 13(c) shows the overhead of enforcing SBRS using optimistic tracking versus HT. 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 HT 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.

6.2.5 *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. Figure 14 shows, for each microbenchmark, the code executed by each of eight threads, as well as runtime overhead over the unmodified JVM. Both microbenchmarks increment a global counter in a loop; syncInc acquires a global lock before every increment, and racyInc never acquires a lock.

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

In contrast, racyInc represents a worst case for HT, since almost all conflicting accesses are involved in data races. HT adds 4,300% overhead, because threads repeatedly trigger coordination to perform pessimistic contended transitions. On further investigation, we find that although only 24% of memory accesses perform pessimistic contended transitions, most of these accesses trigger coordination more than once. HT could alleviate this deficiency by modifying the adaptive policy to transition pessimistic objects back to optimistic states if they trigger coordination frequently.

Pessimistic and optimistic tracking both add about 1,200% 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.

Optimization opportunity. When investigating these performance results in depth, we discovered an optimization opportunity that improves the performance of the optimistic tracking implementation. In particular, releasing a "fat" program lock (Bacon et al. 1998) can incur significant latency. Making a fat lock release operation be a blocking safe point reduces the overhead of optimistic tracking to 22%. This optimization turns many explicit coordination requests to implicit, significantly reducing the runtime overhead of high-conflict programs such as xalan6, avrora9, xalan9, and pjbb2005. Even with this optimization, explicit coordination is still expensive in optimistic tracking. However, the average performance of HT and RT is roughly the same as optimistic tracking with this optimization. On top of this optimization, HT can further improve the performance of several high-conflict programs, and RT is still effective at reducing the cost of explicit coordination. The experiments in this article (and in our prior work (Cao et al. 2016; Zhang et al. 2016)) do *not* include this optimization.

6.3 Evaluating Relaxed Tracking

This section evaluates the performance of RT alone, RT's runtime characteristics, and the performance of relaxed runtime support, compared with optimistic tracking and optimistic runtime support.

6.3.1 Performance of Tracking Alone.

Measuring the problem. We first measure the cost of optimistic tracking, as well as the maximum benefit that can be obtained from optimizations. Figure 15 shows runtime overhead of three configurations over an unmodified JVM. *Optimistic tracking* tracks dependencies strictly and adds 25% overhead on average. Its overhead varies considerably across the evaluated programs, since the overhead is closely linked to the fraction of accesses that trigger coordination using explicit requests, which is the main cost of tracking dependencies.

Ideal is an *unsound* configuration that eliminates most of the cost of strict coordination. In this configuration, after a thread sends an explicit request, it continues execution without waiting for any response. Responding threads in turn ignore requests. (A minor difference between this configuration and the *Ideal* configuration in Figure 11 is that this configuration still performs implicit coordination.) This configuration attempts to estimate an upper bound on the performance that RT might be able to provide. On average, *Ideal* adds 14% overhead—a little more than half of the overhead added by the sound configuration.

The remaining costs are due to fast-path instrumentation at every access, as well as other transitions, including conflicting transitions that trigger coordination using implicit requests. In

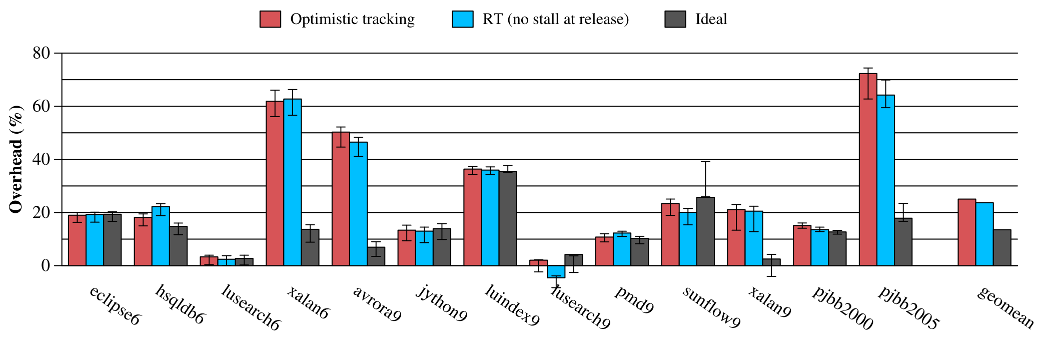


Fig. 15. Runtime overhead added to an unmodified JVM by capturing dependencies using (1) optimistic tracking, compared with (2) RT and (3) an ideal, unsound configuration that eliminates coordination latency.

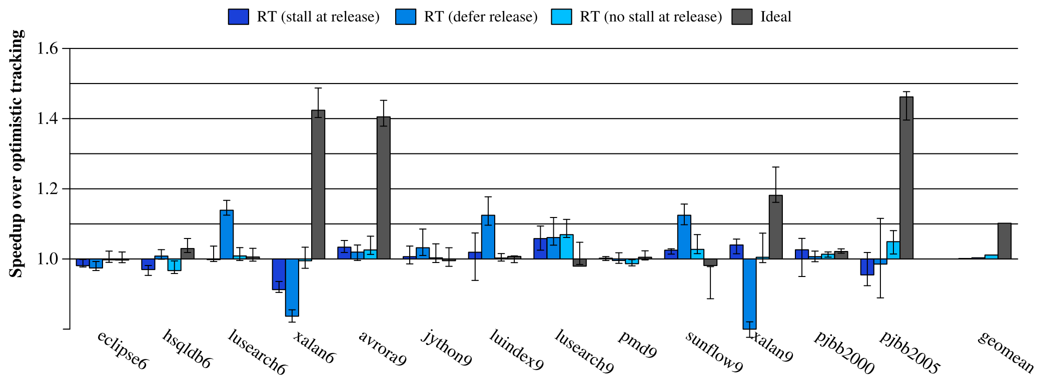


Fig. 16. Speedup of RT relative to optimistic tracking. *Ideal* is an unsound configuration that provides an upper bound on RT's performance.

addition, although requesting threads do not wait for responses and responding threads ignore requests, *Ideal* incurs remote cache misses by sending explicit requests.

RT's effectiveness at hiding coordination costs. The configuration *RT (no stall at release)* in Figure 15 is relaxed tracking using the second optimization from Section 4.1.3. Its average overhead over baseline execution is 24%, a small reduction from optimistic tracking's 25% overhead. As we show later, although RT hides the latency of explicit coordination, it changes the balance of explicit versus implicit coordination and incurs other costs that together cancel out its performance improvement on explicit coordination. We note that RT could still significantly outperform optimistic tracking in some synthetic high-conflict programs that are designed to stress multithreaded execution scenarios (results available in the original RT article (Zhang et al. 2016)).

Figure 16 is a *speedup* graph (higher is better) that shows the same configurations as Figure 15 plus two additional RT configurations. The RT configurations, which are all sound, are as follows:

—*RT (stall at release)* – The default design from Section 4.1. Threads wait at PSROs for all outstanding relaxed stores.
—*RT (defer release)* – The first optimization described in Section 4.1.3. At a lock release, a thread defers the lock release if there are outstanding relaxed stores.

—*RT (no stall at release)* – The second optimization described in Section 4.1.3 (and also shown in Figure 15). At a lock release, a thread continues execution even if there are outstanding relaxed stores. However, no thread except T can read from an object in $\mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Int}}$ state.

The three RT configurations offer minimal average speedup over optimistic tracking (1.01× at the most). They do not help much with the gap between *optimistic tracking* and *Ideal* for the high-conflict programs (hsqldb6, benchxalan6, avrora9, xalan9, and pjbb2005). As we show later, RT changes the balance of explicit versus implicit coordination requests (relative to optimistic tracking) by causing threads to spend more time executing code instead of blocking at safe points. This change cancels out RT's potential performance benefits for these programs, and it represents a challenge for future work. Another significant source of RT overhead is bookkeeping costs: Its queue representation leads to more costs than for optimistic tracking, and it performs additional work to maintain relaxed events.

The *RT (defer release)* configuration does not improve performance on average (nor significantly for any individual program) compared with the *RT (stall at release)*. Although deferring lock release operations has the potential to hide coordination latency, it incurs two additional costs. First, deferring releases incurs additional bookkeeping costs. Second, deferring releases often changes the balance between explicit and implicit requests triggered for coordination, since threads are more likely to be executing code rather than blocked at release operations waiting for coordination responses. These factors are enough to outweigh any potential benefit provided by deferring releases.

Similarly, the *RT (no stall at release)* configuration helps hide latency, but it introduces another source of latency: Any thread other than T must wait to read an object locked in $\mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Int}}$ state. On average, these factors cancel each other, so *RT (no stall at release)* provides almost no average benefit over *RT (stall at release)*.

6.3.2 Runtime characteristics. Next we focus on understanding factors contributing to the performance results. Table 3 reports runtime statistics for *RT (no stall at release)* and optimistic tracking.

For each type of tracking, *State transitions* counts how many accesses execute instrumentation that requires either no state change (*Same state*) or a *Conflicting* transition that triggers coordination. An interesting phenomenon is that RT sometimes reduces *how many* conflicting transitions occur, relative to optimistic tracking. This phenomenon occurs because of cases in which an object is highly contended, and two or more threads transfer its ownership in quick succession. When using optimistic tracking, a thread must wait for coordination at each access, enabling another thread to make progress and trigger coordination for the next access to the object, leading to many conflicting transitions. In contrast, when using RT—particularly when executing past release operations as permitted by the *RT no stall at release* configuration—a thread is more likely to perform consecutive *relaxed* accesses to a contended object, leading to fewer conflicting transitions, compared with optimistic tracking.

The *Coord. requests* columns count explicit and implicit requests, which can sum to more than *Conflicting* transitions, because RdSh-to-WrEx transitions involve multiple requests. Programs with more explicit requests generally have higher coordination overhead and can benefit more from RT.

The *Coord. responses* columns tally RT responses. Each sum equals the number of explicit requests, since there is one response for every explicit request. Since explicit responses do not incur latency, the ratio of explicit to implicit responses does not affect performance significantly.

The last two columns count relaxed accesses. While some of these accesses occur immediately after coordination requests in a thread's execution, others are repeat accesses to the same memory

Table 3. Runtime Characteristics of Optimistic (i.e., Strict) Tracking and Relaxed Tracking

	Optimistictracking				**Relaxed tracking**							
	State transitions		**Coord. requests**		**State transitions**		**Coord. requests**		**Coord. responses**		**Relaxed accesses**	
	Same state	Conflicting	Explicit	Implicit	Same state	Conflicting	Explicit	Implicit	Explicit	Implicit	Read	Write
eclipse6	1.2×10^{10}	$1.4 \times 10^5 (<0.01\%)$	1.6×10^4	2.9×10^5	1.2×10^{10}	$1.4 \times 10^5 (<0.01\%)$	1.1×10^4	2.6×10^5	9.6×10^3	1.4×10^3	1.4×10^4	4.8×10^3
hsqldb6	6.2×10^8	$9.0 \times 10^5 (0.14\%)$	3.5×10^4	3.8×10^6	6.2×10^8	$9.0 \times 10^5 (0.14\%)$	3.4×10^4	4.4×10^6	3.1×10^4	3.5×10^3	4.7×10^4	3.8×10^4
lusearch6	2.4×10^9	$4.4 \times 10^3 (<0.01\%)$	2.3×10^3	4.5×10^3	2.4×10^9	$4.4 \times 10^3 (<0.01\%)$	2.3×10^3	4.6×10^3	8.8×10^2	1.4×10^3	2.2×10^3	2.1×10^3
xalan6	1.1×10^{10}	$1.9 \times 10^7 (0.17\%)$	1.3×10^7	5.9×10^6	1.1×10^{10}	$1.9 \times 10^7 (0.17\%)$	1.4×10^7	5.2×10^6	1.3×10^7	6.3×10^5	1.6×10^7	1.8×10^7
avrora9	6.1×10^9	$5.9 \times 10^6 (0.097\%)$	4.1×10^6	1.8×10^7	6.1×10^9	$5.7 \times 10^6 (0.093\%)$	2.8×10^6	1.4×10^7	2.2×10^6	5.5×10^5	2.1×10^6	1.9×10^6
jython9	5.1×10^9	$6.6 \times 10^1 (<0.01\%)$	1.8×10^1	1.5×10^0	5.1×10^9	$6.2 \times 10^1 (<0.01\%)$	1.5×10^1	4.5×10^0	1.3×10^1	2.0×10^0	2.3×10^1	0
luindex9	3.5×10^8	$3.7 \times 10^2 (<0.01\%)$	1.5×10^1	3.3×10^2	3.5×10^8	$3.7 \times 10^2 (<0.01\%)$	1.2×10^1	3.3×10^2	9.5×10^0	3.0×10^0	1.9×10^1	2.5×10^0
lusearch9	2.4×10^9	$2.9 \times 10^3 (<0.01\%)$	4.6×10^3	4.4×10^3	2.4×10^9	$2.9 \times 10^3 (<0.01\%)$	5.0×10^3	3.3×10^3	1.3×10^3	3.7×10^3	6.4×10^3	4.1×10^2
pmd9	5.7×10^8	$4.4 \times 10^4 (<0.01\%)$	3.1×10^4	5.3×10^4	5.7×10^8	$4.3 \times 10^4 (<0.01\%)$	2.7×10^4	4.9×10^4	2.0×10^4	6.9×10^3	2.5×10^4	3.4×10^4
sunflow9	1.7×10^{10}	$1.4 \times 10^4 (<0.01\%)$	1.5×10^4	7.6×10^3	1.7×10^{10}	$9.3 \times 10^3 (<0.01\%)$	9.6×10^3	8.7×10^3	3.3×10^3	6.3×10^3	2.4×10^5	8.4×10^3
xalan9	1.0×10^{10}	$1.8 \times 10^7 (0.18\%)$	9.7×10^6	8.7×10^6	1.0×10^{10}	$2.0 \times 10^7 (0.20\%)$	1.3×10^7	7.1×10^6	1.3×10^7	6.4×10^5	2.0×10^7	2.1×10^7
pjbb2000	1.7×10^9	$9.5 \times 10^5 (0.055\%)$	6.2×10^4	9.0×10^5	1.7×10^9	$9.5 \times 10^5 (0.055\%)$	6.1×10^4	9.0×10^5	5.7×10^4	3.5×10^3	2.3×10^5	9.7×10^4
pjbb2005	6.6×10^9	$4.6 \times 10^7 (0.69\%)$	3.2×10^7	5.7×10^7	6.5×10^9	$4.1 \times 10^7 (0.61\%)$	2.5×10^7	6.2×10^7	1.9×10^7	5.5×10^6	1.2×10^7	1.6×10^7

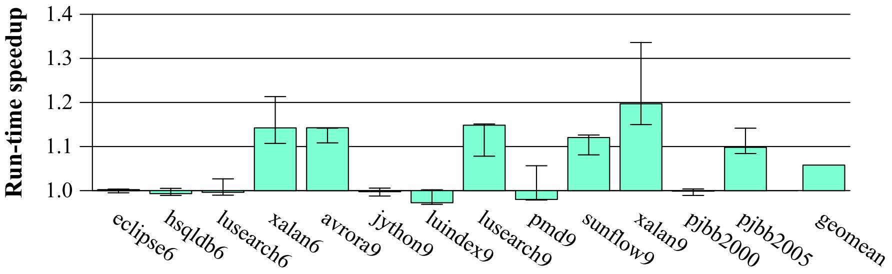


Fig. 17. runtime speedup of the relaxed dependence recorder over the optimistic dependence recorder.

location or loads from objects for which some *other* thread has initiated coordination. Due to these cases, relaxed accesses can outnumber conflicting transitions. On the other hand, conflicting transitions can outnumber relaxed accesses, since an implicit request does not lead to a relaxed access.

6.3.3 Performance of Runtime Support. This section evaluates whether RT can benefit runtime support that detects cross-thread dependencies (dependence recorder) and controls cross-thread dependencies (STM).

Dependence recorder. Figure 17 shows the performance of the optimistic and relaxed recorders. Not surprisingly, the performance story for the recorders is similar to the story for tracking dependencies alone. On average, the RT recorder is 1.06× faster than the optimistic recorder.

We note that although RT can hide coordination latency, the relaxed recorder still needs to record each happens-before edge. Some relaxed loads can avoid conflicting transitions and coordination entirely, but the recorder must log each of the loaded values. The relaxed recorder thus often logs *more* than the optimistic recorder. Notably, the relaxed recorder's log size is about 2× the optimistic recorder's for lusearch6, xalan6, and xalan9 (results not shown). For all other programs, the relaxed recorder logs less than 50% more than the optimistic recorder.

Software transactional memory. Next we compare the optimistic and relaxed STMs using the transactional STAMP benchmarks. Figure 18 shows the execution time of the optimistic and relaxed STMs. We first note that both STMs typically scale poorly after eight threads: Our platform has 8 cores per socket, leading to greater inter-thread communication for more than eight threads, and prior work has found that STAMP has limited scalability (Zyulkyarov et al. 2010).

For genome and vacation, RT reduces overhead for two to eight application threads, but the benefit decreases with more threads. For genome, the ratio of implicit to explicit requests increases substantially with more threads (statistics not shown), leading to fewer opportunities for RT to improve performance. For vacation, the implicit-to-explicit ratio stays fairly constant across thread counts, but RT's benefit diminishes, because accesses per thread decrease as threads increase, leading to less latency per thread for RT to reduce.

Fewer than 0.01% of labyrinth3d's accesses trigger coordination, so it cannot benefit noticeably from RT.

For kmeans, intruder, and ssca2, RT provides sustained or increasing benefit over optimistic tracking for 8 to 32 threads. For these programs, the relaxed STM achieves a significantly lower rate of aborted transactions than the optimistic STM. The relaxed STM validates relaxed loads at object field and array element granularity, as opposed to the optimistic STM's loads, which use reader locks and read sets at object granularity, leading to more transactional conflicts due to false sharing—an interesting side effect of supporting relaxed loads. However, direct benefits from RT are limited, because many transactions are short, and the relaxed STM must wait at transaction end for all outstanding relaxed accesses (Section 4.3.2).

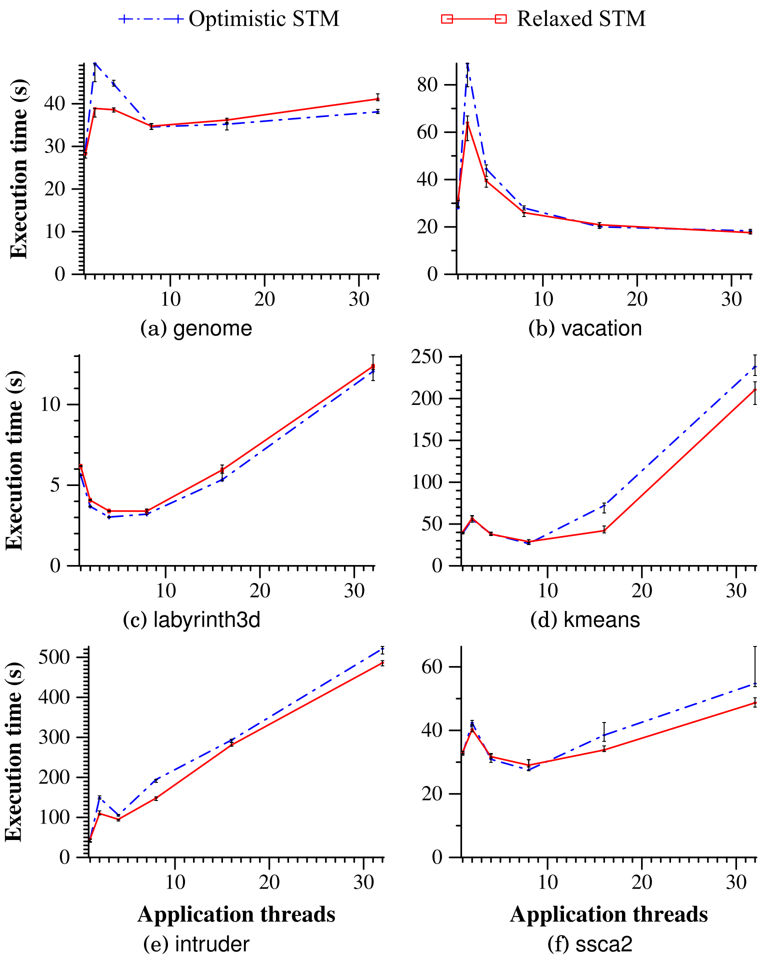


Fig. 18. Performance of the optimistic and relaxed STMs.

In summary, RT reduces the cost of tracking dependencies for programs with high coordination costs, but its benefit is limited by correctness constraints (e.g., limitations on deferring stores past synchronization) and indirect effects (e.g., increases to the explicit-to-implicit request ratio when using RT compared with optimistic tracking). Although the relaxed recorder and STM suffer drawbacks (increased log size and transactional correctness constraints, respectively) that limit the improvement somewhat, these results demonstrate the potential of RT to improve the performance of dependence-tracking-based runtime support.

7 RELATED WORK

This section compares with prior work not covered already.

Program locks. This article 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 (Russell and Detlefs 2006;

Kawachiya et al. 2002; Burrows 2004; Vasudevan et al. 2010). 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 STM for enforcing atomicity of critical sections (Usui et al. 2009). 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 (Abadi et al. 2009). Dice et al. build a runtime library that supports adaptive lock elision using hardware transactional memory (HTM) and optimistic software execution (Dice et al. 2014). Ziv et al. formalize a theory for correctly composing different concurrency control protocols in programs (Ziv et al. 2015).

Von Praun and Gross introduce a data race detection analysis that uses an optimistic (biased) for non-shared objects and a pessimistic (unbiased) approach for shared objects (von Praun and Gross 2001). However, their approach adds high overhead when shared objects are accessed frequently (Bond et al. 2013).

Neamtiu and Hicks introduce "relaxed synchronization" to allow threads to keep executing while waiting to join a global synchronization barrier (for dynamic software update) (Neamtiu and Hicks 2009). However, their work is not applicable to tracking dependencies.

Tracking dependencies using commodity hardware. Intel processors now provide best-effort HTM (Yoo et al. 2013), which could potentially help track dependencies efficiently. However, recent work suggests that commodity HTM struggles to outperform software-based synchronization if transactions are short (Matar et al. 2014; Ritson and Barnes 2013; Yoo et al. 2013; Sengupta et al. 2017). Liu et al. show how to record and replay commodity HTM transactions, but the approach does not support recording and replaying programs in general (Liu et al. 2015).

8 CONCLUSION

This article presents two distinct, software-only approaches that tackle the performance limitations of existing dependence tracking mechanisms. HT uses a hybrid state model and adaptive policy to combine pessimistic and optimistic tracking and outperform both individually. RT reduces optimistic tracking's coordination latency by relaxing the requirement of tracking all dependencies soundly, allowing coordination latency to overlap with useful program work. We build runtime support based on HT and RT to demonstrate their potentials. The results show the benefits—and challenges and limitations—of applying HT and RT to real runtime systems and real applications. This work advances the state of art in dependence tracking, illuminating future directions for efficient, software-only parallel runtime support that targets diverse applications.

APPENDIXES

A HYBRID TRACKING'S INSTRUMENTATION PSEUDOCODE

Figure 19 shows the instrumentation added by HT. For simplicity, we only show instrumentation for a program store. The instrumentation for loads is more complex, because it also handles RdEx_T^* and RdSh_c^* states and supports reentrant reader locks.

The fast path (Figure 19(a)) only checks for the $\mathsf{WrEx}_T^{\mathsf{Opt}}$ state, since we expect that the majority of accesses trigger same-state optimistic transitions. The slow path (Figure 19(c)) changes the state based on HT's state transitions (Figure 4 and Table 4). The slow path repeatedly reloads and tries to change the state if an atomic update fails. A contended transition triggers coordination (line 44); then the slow path retries until the state becomes unlocked, enabling an uncontended transition

```
22 if (o.state != WrEx$_T^{Opt}$) {
23   slowPath(o);
24 }
25 o.f = ... ; // program store
```

(a) The instrumentation fast path for HT.

```
26 for (o : T.lockBuffer) {
27   o.state = AdaptivePolicy.toOpt(o) ?
28                     WrEx$_T^{Opt}$ : WrEx$_T^{Pess}$;
29 }
```

(b) Instrumentation at PSROs and responding safe points.

```
30 slowPath(o) {
31   while(true) {
32     state = o.state;
33     if (isPess(state)) { // Pessimistic
34       // Pessimistic Locked, uncontended
35       if (state == WrEx$_T^{WLock}$) break;
36       // Pessimistic Unlocked
37       if (isUnlocked(state) ||
38           state == WrEx$_T^{RLock}$) {
39         if (CAS(&o.state, state, WrEx$_T^{WLock}$)) {
40           T.lockBuffer.add(o);
41           break;
42         }
43       } else { // Pessimistic Locked, contended
44         coordinate(getOwner(state));
45       }
46     } else { // Optimistic
47       if (state == RdEx$_T^{Opt}$) { ... }
48       if ((state != WrEx$_*^{Int}$) &&
49           CAS(&o.state, state, WrEx$_T^{Int}$)) {
50         coordinate(getOwner(state));
51         // Decision from adaptive policy
52         if (AdaptivePolicy.toPess(o)) {
53           o.state = WrEx$_T^{WLock}$;
54           T.lockBuffer.add(o);
55         } else {
56           o.state = WrEx$_T^{Opt}$;
57         }
58         break;
59       }
60     }
61     /* blocking safe point */
62   }
63 }
```

(c) The instrumentation slow path for HT.

Fig. 19. Instrumentation added by HT for program stores only. (Handling loads is analogous but more complex.)

(lines 37–42). On a successful transition to a pessimistic state, the instrumentation adds the object to the per-thread lock buffer (lines 40 and 54).

Figure 19(b) 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 3.4). The pseudocode shows how to handle objects in WrEx$_T^{WLock}$ state only, not other states.

B HYBRID TRACKING'S COMPLETE STATE TRANSITIONS

Table 4 shows all possible transitions for HT's hybrid state model. 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 PSROs.

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 ∉ T.rdSet), it adds the object to its read set: T.rdSet ← T.rdSet ∪ {o}. Whenever T flushes its lock buffer, it also clears its read set: T.rdSet ← ∅.

Table 4. All Possible State Transitions for the Hybrid State Model

Trans. type	Old state	Program access	New state	Sync. needed	Cross-thread dependence?
Pessimistic uncontended (reentrant)	$\mathrm{WrEx}_T^{\mathrm{WLock}}$	R or W by T	Same	None	No
	$\mathrm{WrEx}_T^{\mathrm{RLock}}$	R by T			
	$\mathrm{RdEx}_T^{\mathrm{RLock}}$	R by T			
	$\mathrm{RdSh}_c^{\mathrm{RLock(n)}}$	R by T if $o \in$ T.rdSet			
Pessimistic uncontended	$\mathrm{WrEx}_T^{\mathrm{Pess}}$	W by T	$\mathrm{WrEx}_T^{\mathrm{WLock}}$	CAS	No
	$\mathrm{WrEx}_T^{\mathrm{Pess}}$	R by T	$\mathrm{WrEx}_T^{\mathrm{RLock}}$		
	$\mathrm{RdEx}_T^{\mathrm{Pess}}$	R by T	$\mathrm{RdEx}_T^{\mathrm{RLock}}$		
	$\mathrm{RdEx}_T^{\mathrm{Pess}}$ or $\mathrm{RdEx}_T^{\mathrm{RLock}}$ or $\mathrm{WrEx}_T^{\mathrm{RLock}}$	W by T	$\mathrm{WrEx}_T^{\mathrm{WLock}}$	CAS	No
	$\mathrm{RdEx}_{T1}^{\mathrm{Pess}}$	R by T2	$\mathrm{RdSh}_{\mathrm{gRdShCount}}^{\mathrm{RLock(1)}}$		maybe
	$\mathrm{RdEx}_{T1}^{\mathrm{RLock}}$ or $\mathrm{WrEx}_{T1}^{\mathrm{RLock}}$	R by T2	$\mathrm{RdSh}_{\mathrm{gRdShCount}}^{\mathrm{RLock(2)}}$		maybe
	$\mathrm{RdSh}_c^{\mathrm{Pess}}$	R by T	$\mathrm{RdSh}_c^{\mathrm{RLock(1)}*}$	CAS	maybe
	$\mathrm{RdSh}_c^{\mathrm{RLock(n)}}$	R by T if $o \notin$ T.rdSet	$\mathrm{RdSh}_c^{\mathrm{RLock(n+1)}*}$		
	$\mathrm{WrEx}_{T1}^{\mathrm{Pess}}$	W by T2	$\mathrm{WrEx}_{T2}^{\mathrm{WLock}}$	CAS	maybe
	$\mathrm{WrEx}_{T1}^{\mathrm{Pess}}$	R by T2	$\mathrm{RdEx}_{T2}^{\mathrm{RLock}}$		
	$\mathrm{RdEx}_{T1}^{\mathrm{Pess}}$	W by T2	$\mathrm{WrEx}_{T2}^{\mathrm{WLock}}$		
	$\mathrm{RdSh}_c^{\mathrm{Pess}}$	W by T	$\mathrm{WrEx}_T^{\mathrm{WLock}}$		
Pessimistic contended	$\mathrm{WrEx}_{T1}^{\mathrm{WLock}}$	W by T2	Handled at owner thread(s)' responding safe points	Roundtrip coordination	maybe
	$\mathrm{WrEx}_{T1}^{\mathrm{RLock}}$	W by T2			
	$\mathrm{WrEx}_{T1}^{\mathrm{WLock}}$	R by T2			
	$\mathrm{RdEx}_{T1}^{\mathrm{RLock}}$	W by T2			
	$\mathrm{RdSh}_c^{\mathrm{RLock(n)}}$	W by T2			

(Continued)

Table 4. Continued

Trans. type	Old state	Program access	New state	Sync. needed	Cross-thread dependence?
Pessimistic unlock **OR** Pess → Opt	$\text{WrEx}_T^{\text{WLock}}$ or $\text{WrEx}_T^{\text{RLock}}$	PSRO or responding safe point	$\text{WrEx}_T^{\text{Pess}}$ **OR** $\text{WrEx}_T^{\text{Opt}}$	CAS	N/A
	$\text{RdEx}_T^{\text{RLock}}$		$\text{RdEx}_T^{\text{Pess}}$ **OR** $\text{RdEx}_T^{\text{Opt}}$		
	$\text{RdSh}_c^{\text{RLock(n)}}$ if $n > 1$		$\text{RdSh}_c^{\text{RLock(n-1)}}$		
	$\text{RdSh}_c^{\text{RLock(1)}}$		$\text{RdSh}_c^{\text{Pess}}$ **OR** $\text{RdSh}_c^{\text{Opt}}$		
Same state	$\text{WrEx}_T^{\text{Opt}}$	R or W by T	Same	None	No
	$\text{RdEx}_T^{\text{Opt}}$	R by T			
	$\text{RdSh}_c^{\text{Opt}}$	R by T if $\text{T.rdShCount} \geq c$			
Upgrading	$\text{RdEx}_T^{\text{Opt}}$	W by T	$\text{WrEx}_T^{\text{Opt}}$	CAS	No
	$\text{RdEx}_{T1}^{\text{Opt}}$	R by T2	$\text{RdSh}_{\text{gRdShCount}}^{\text{Opt}}$		maybe
Fence	$\text{RdSh}_c^{\text{Opt}}$	R by T if $\text{T.rdShCount} < c$	$(\text{T.rdShCount} \leftarrow c)$	Memory fence	maybe
Conflicting **OR** Opt → Pess	$\text{WrEx}_{T1}^{\text{Opt}}$	W by T2	$\text{WrEx}_{T2}^{\text{Int}} \rightarrow \text{WrEx}_{T2}^{\text{Opt}}$ **OR** $\text{WrEx}_{T2}^{\text{WLock}}$	Roundtrip coordination	maybe
	$\text{WrEx}_{T1}^{\text{Opt}}$	R by T2	$\text{RdEx}_{T2}^{\text{Int}} \rightarrow \text{RdEx}_{T2}^{\text{Opt}}$ **OR** $\text{RdEx}_{T2}^{\text{RLock}}$		
	$\text{RdEx}_{T1}^{\text{Opt}}$	W by T2	$\text{WrEx}_{T2}^{\text{Int}} \rightarrow \text{WrEx}_{T2}^{\text{Opt}}$ **OR** $\text{WrEx}_{T2}^{\text{WLock}}$		
	$\text{RdSh}_c^{\text{Opt}}$	W by T2	$\text{WrEx}_{T2}^{\text{Int}} \rightarrow \text{WrEx}_{T2}^{\text{Opt}}$ **OR** $\text{WrEx}_{T2}^{\text{WLock}}$		

Note: Instances of "**OR**" indicate cases in which a state can potentially transition between pessimistic and optimistic states.
*Pessimistic uncontended transitions from RdSh_c^* to $\text{RdSh}_c^{\text{RLock(*)}}$ also update T.rdShCount to max(T.rdShCount, c).

ACKNOWLEDGMENTS

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

REFERENCES

Martín Abadi, Tim Harris, and Mojtaba Mehrara. 2009. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *PPoPP*. 185–196. DOI: http://dx.doi.org/10.1145/1504176.1504203

Sarita V. Adve and Hans-J. Boehm. 2010. Memory models: A case for rethinking parallel languages and hardware. *Commun. ACM* 53, 8 (2010), 90–101. DOI: http://dx.doi.org/10.1145/1787234.1787255

Sarita V. Adve and Mark D. Hill. 1990. Weak ordering—a new definition. In *ISCA*. 2–14. DOI: http://dx.doi.org/10.1145/325164.325100

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, Susan Flynn 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. 2000. The Jalapeño virtual machine. *IBM Syst. J.* 39, 1 (2000), 211–238.

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. 2005. The Jikes research virtual machine project: Building an open-source research community. *IBM Syst. J.* 44, 2 (2005), 399–417. DOI: http://dx.doi.org/10.1147/sj.442.0399

David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. 1998. Thin locks: Featherweight synchronization for java. In *PLDI*. 258–268. DOI: http://dx.doi.org/10.1145/277650.277734

Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. 2015. Valor: Efficient, software-only region conflict exceptions. In *OOPSLA*. 241–259. DOI: http://dx.doi.org/10.1145/2814270.2814292

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. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*. 169–190.

Hans-J. Boehm. 2012. Position paper: Nondeterminism is unavoidable, but data races are pure evil. In *RACES*. 9–14.

Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. In *PLDI*. 68–78. DOI: http://dx.doi.org/10.1145/1375581.1375591

Michael D. Bond, Milind Kulkarni, Man Cao, Meisam Fathi Salmi, and Jipeng Huang. 2015. Efficient deterministic replay of multithreaded executions in a managed language virtual machine. In *PPPJ*. 90–101.

Michael D. Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. 2013. Octet: Capturing and controlling cross-thread dependences efficiently. In *OOPSLA*. 693–712.

Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*. 211–230. DOI: http://dx.doi.org/10.1145/582419.582440

Mike Burrows. 2004. How to implement unnecessary mutexes. In *Computer Systems Theory, Technology, and Applications*. Springer–Verlag, 51–57.

Man Cao, Minjia Zhang, and Michael D. Bond. 2014. Drinking from both glasses: Adaptively combining pessimistic and optimistic synchronization for efficient parallel runtime support. In *WoDet*.

Man Cao, Minjia Zhang, Aritra Sengupta, and Michael D. Bond. 2016. Drinking from both glasses: Combining pessimistic and optimistic tracking of cross-thread dependences. In *PPoPP*. Article 20, 13 pages. DOI: http://dx.doi.org/10.1145/2851141.2851143

Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *IISWC*.

Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*. 258–269. DOI: http://dx.doi.org/10.1145/512529.512560

Luke Dalessandro and Michael L. Scott. 2012. Sandboxing transactional memory. In *PACT*. 171–180. DOI: http://dx.doi.org/10.1145/2370816.2370843

Luke Dalessandro, Michael F. Spear, and Michael L. Scott. 2010. NOrec: Streamlining STM by abolishing ownership records. In *PPoPP*. 67–78. DOI: http://dx.doi.org/10.1145/1693453.1693464

Brian Demsky and Alokika Dash. 2010. Evaluating contention management using discrete event simulation. In *TRANSACT*.

Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. 2014. Adaptive integration of hardware and software lock elision techniques. In *SPAA*. 188–197. DOI: http://dx.doi.org/10.1145/2612669.2612696

Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. 2009. Stretching transactional memory. In *PLDI*. 155–165. DOI: http://dx.doi.org/10.1145/1542476.1542494

Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: A race and transaction-aware java runtime. In *PLDI*. 245–255. DOI: http://dx.doi.org/10.1145/1250734.1250762

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and precise dynamic race detection. In *PLDI*. 121–133. DOI: http://dx.doi.org/10.1145/1542476.1542490

Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. 2008. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*. 293–303. DOI: http://dx.doi.org/10.1145/1375581.1375618

Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. 2004. Transactional memory coherence and consistency. In *ISCA*. 102–113.

Tim Harris and Keir Fraser. 2003. Language support for lightweight transactions. In *OOPSLA*. 388–402. DOI: http://dx.doi.org/10.1145/949305.949340

Tim Harris, James Larus, and Ravi Rajwar. 2010. *Transactional Memory* (2nd ed.). Morgan & Claypool Publishers.

Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. 2006. Optimizing memory transactions. In *PLDI*. 14–25. DOI: http://dx.doi.org/10.1145/1133981.1133984

Yanyan Jiang, Du Li, Chang Xu, Xiaoxing Ma, and Jian Lu. 2015. Optimistic shared memory dependence tracing. In *ASE*. 524–534. DOI: http://dx.doi.org/10.1109/ASE.2015.11

Tomas Kalibera, Matthew Mole, Richard Jones, and Jan Vitek. 2012. A black-box approach to understanding concurrency in DaCapo. In *OOPSLA*. 335–354. DOI: http://dx.doi.org/10.1145/2384616.2384641

Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. 2002. Lock reservation: Java locks can mostly do without atomic operations. In *OOPSLA*. 130–141. DOI: http://dx.doi.org/10.1145/582419.582433

Guy Korland, Nir Shavit, and Pascal Felber. 2010. Deuce: Noninvasive software transactional memory in java. *Trans. High Perf. EAC* 5, 2 (2010).

Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565. DOI: http://dx.doi.org/10.1145/359545.359563

T. J. LeBlanc and J. M. Mellor-Crummey. 1987. Debugging parallel programs with instant replay. *IEEE Trans. Comput.* 36, 4 (1987), 471–482. DOI: http://dx.doi.org/10.1109/TC.1987.1676929

Dongyoon Lee, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2012. Chimera: Hybrid program analysis for determinism. In *PLDI*. 463–474. DOI: http://dx.doi.org/10.1145/2254064.2254119

Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. 2010. Respec: Efficient online multiprocessor replay via speculation and external determinism. In *ASPLOS*. 77–90. DOI: http://dx.doi.org/10.1145/1736020.1736031

Yujie Liu, Justin Gottschlich, Gilles Pokam, and Michael Spear. 2015. TSXProf: Profiling hardware transactions. In *PACT*. 75–86. DOI: http://dx.doi.org/10.1109/PACT.2015.28

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The java memory model. In *POPL*. 378–391. DOI: http://dx.doi.org/10.1145/1040305.1040336

Hassan Salehe Matar, Ismail Kuru, Serdar Tasiran, and Roman Dementiev. 2014. Accelerating precise race detection using commercially-available hardware transactional memory support. In *WoDet*.

Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. 2008. Practical weak-atomicity semantics for java STM. In *SPAA*. 314–325. DOI: http://dx.doi.org/10.1145/1378533.1378588

Iulian Neamtiu and Michael Hicks. 2009. Safe and timely updates to multi-threaded programs. In *PLDI*. 13–24. DOI: http://dx.doi.org/10.1145/1542476.1542479

Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. 2008. Design and implementation of transactional constructs for C/C++. In *OOPSLA*. 195–212. DOI: http://dx.doi.org/10.1145/1449764.1449780

Marek Olszewski, Jeremy Cutler, and J. Gregory Steffan. 2007. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *PACT*. 365–375. DOI: http://dx.doi.org/10.1109/PACT.2007.42

Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2013. ...and region serializability for all. In *HotPar*.

Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. 2009. PRES: Probabilistic replay with execution sketching on multiprocessors. In *SOSP*. 177–192. DOI: http://dx.doi.org/10.1145/1629575.1629593

Carl G. Ritson and Frederick R. M. Barnes. 2013. An evaluation of Intel's restricted transactional memory for CPAs. In *CPA*. 271–292.

Michiel Ronsse and Koen De Bosschere. 1999. RecPlay: A fully integrated practical record/replay system. *Trans. Comput. Syst.* 17, 2 (1999), 133–152. DOI: http://dx.doi.org/10.1145/312203.312214

Kenneth Russell and David Detlefs. 2006. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *OOPSLA*. 263–272. DOI: http://dx.doi.org/10.1145/1167473.1167496

Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. 2006. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP*. 187–197. DOI:http://dx.doi.org/10.1145/1122971.1123001

Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. 1996. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *ASPLOS*. 174–185. DOI:http://dx.doi.org/10.1145/237090.237179

Aritra Sengupta, Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Milind Kulkarni. 2015. Hybrid static–dynamic analysis for statically bounded region serializability. In *ASPLOS*. 561–575. DOI:http://dx.doi.org/10.1145/2694344.2694379

Aritra Sengupta, Man Cao, Michael D. Bond, and Milind Kulkarni. 2017. Legato: End-to-end bounded region serializability using commodity hardware transactional memory. In *CGO*. 1–13.

Nehir Sonmez, Tim Harris, Adrian Cristal, Osman S. Unsal, and Mateo Valero. 2009. Taking the heat off transactions: Dynamic selection of pessimistic concurrency control. In *IPDPS*. 1–10. DOI:http://dx.doi.org/10.1109/IPDPS.2009.5161032

Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. 2009. A comprehensive strategy for contention management in software transactional memory. In *PPoPP*. 141–150. DOI:http://dx.doi.org/10.1145/1504176.1504199

Takayuki Usui, Reimer Behrends, Jacob Evans, and Yannis Smaragdakis. 2009. Adaptive locks: Combining transactions and locks for efficient concurrency. In *PACT*. 3–14. DOI:http://dx.doi.org/10.1109/PACT.2009.20

Nalini Vasudevan, Kedar S. Namjoshi, and Stephen A. Edwards. 2010. Simple and fast biased locks. In *PACT*. 65–74. DOI:http://dx.doi.org/10.1145/1854273.1854287

Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. DoublePlay: Parallelizing sequential logging and replay. In *ASPLOS*. 15–26. DOI:http://dx.doi.org/10.1145/1950365.1950370

Christoph von Praun and Thomas R. Gross. 2001. Object race detection. In *OOPSLA*. 70–82. DOI:http://dx.doi.org/10.1145/504282.504288

Christoph von Praun and Thomas R. Gross. 2003. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI*. 115–128. DOI:http://dx.doi.org/10.1145/781131.781145

Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. 2010. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *ASPLOS*. 155–166. DOI:http://dx.doi.org/10.1145/1736020.1736039

Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance evaluation of intel transactional synchronization extensions for high-performance computing. In *SC*. Article 19, 11 pages. DOI:http://dx.doi.org/10.1145/2503210.2503232

Minjia Zhang, Swarnendu Biswas, and Michael D. Bond. 2016. Relaxed dependence tracking for parallel runtime support. In *CC'16*. 45–55. DOI:http://dx.doi.org/10.1145/2892208.2892229

Minjia Zhang, Jipeng Huang, Man Cao, and Michael D. Bond. 2015. Low-overhead software transactional memory with progress guarantees and strong semantics. In *PPoPP*. 97–108. DOI:http://dx.doi.org/10.1145/2688500.2688510

Ofri Ziv, Alex Aiken, Guy Golan-Gueta, G. Ramalingam, and Mooly Sagiv. 2015. Composing concurrency control. In *PLDI*. 240–249. DOI:http://dx.doi.org/10.1145/2737924.2737970

Ferad Zyulkyarov, Srdjan Stipic, Tim Harris, Osman S. Unsal, Adrián Cristal, Ibrahim Hur, and Mateo Valero. 2010. Discovering and understanding performance bottlenecks in transactional applications. In *PACT*. 285–294. DOI:http://dx.doi.org/10.1145/1854273.1854311

Received December 2016; revised April 2017; accepted May 2017