Relaxed Dependence Tracking for Parallel Runtime Support *

Minjia Zhang Swarnendu Biswas Michael D. Bond
Ohio State University (USA)
{zhanminj,biswass,mikebond}@cse.ohio-state.edu

Abstract

It is notoriously difficult to achieve both correctness and scalability for many shared-memory parallel programs. To improve correctness and scalability, researchers have developed various kinds of *parallel runtime support* such as multithreaded record & replay and software transactional memory. Existing forms of runtime support slow programs significantly in order to track an execution's cross-thread dependences accurately.

This paper investigates the potential for runtime support to hide latency introduced by dependence tracking, by tracking dependences in a *relaxed* way—meaning that not all dependences are tracked accurately. The key challenge in relaxing dependence tracking is to preserve both the program's semantics and the runtime support's guarantees. We present an approach called *relaxed dependence tracking* (RT) and demonstrate its potential by building two types of RT-based runtime support. Our evaluation shows that RT hides much of the latency incurred by dependence tracking, although RT-based runtime support incurs costs and complexity in order to handle relaxed dependence information. By demonstrating how to relax dependence tracking to hide latency while preserving correctness, this work shows the potential for addressing a key cost of dependence tracking, thus advancing knowledge in the design of parallel runtime support.

Categories and Subject Descriptors D.1.4 [*Programming Languages*]: Processors—Run-time environments

Keywords Runtime support for parallelism, dependence tracking, multithreaded record & replay, software transactional memory

1. Introduction

Software must become more parallel in order to fully utilize hardware that provides more, instead of faster, cores in successive generations. However, achieving both correctness and scalability for shared-memory parallel programs is notoriously difficult. To this end, researchers and practitioners have developed various kinds of *parallel runtime support* that check or enforce concurrency correctness properties such as atomicity, data-race freedom, and determinism. For example, *multithreaded record & replay* enables offline debugging and online replication [18, 26, 27, 36, 52, 56, 60]. *Transactional memory* enforces atomicity, avoiding several challenges associated with using fine-grained locking [13, 21, 28, 30, 31, 33, 40, 41, 47, 51, 59, 62]. However, existing runtime support is *impractical*: it relies on unrealistic custom hardware, adds high overhead, or has other serious limitations. This paper focuses on runtime support for commodity systems, often called "software only." Runtime support generally needs to *track* (i.e., detect or control) cross-thread data dependences soundly, which entails using *synchronized* instrumentation to ensure that each program access and its instrumentation execute together atomically (Section 2.1).

This paper explores the potential for reducing the run-time overhead of tracking dependences by *relaxing the requirement that runtime support must track all dependences accurately*—while preserving the runtime support's guarantees and adhering to the language's semantics. We introduce *relaxed dependence tracking* (RT), which enables a thread to continue executing past a memory access involved in a cross-thread dependence, without accurately tracking the dependence. Our design of RT targets dependence tracking based on so-called *biased reader–writer locking* [11, 14, 31, 32, 46, 48, 49, 57] (Section 2.2), which avoids the costs of reacquiring a lock for non-conflicting accesses, but incurs latency at conflicting accesses in order to perform *coordination* among conflicting threads (Section 2.3). The high cost of coordination provides both a challenge and an opportunity for RT to hide this latency, by relaxing the tracking of dependences at accesses involved in dependences. RT's design consists of two elements: a *relaxed coordination protocol* and support for *relaxed loads and stores*. We show that in order to overlap relaxed loads and stores with relaxed coordination correctly, relaxed stores should be deferred, and relaxed loads should be logged and handled in a runtime-support-specific way.

In addition to designing RT, we design two kinds of runtime support that use RT: (1) an *RT-based dependence recorder* and (2) an *RT-based software transactional memory (STM) system*. We implement RT and the RT-based recorder and STM in a high-performance Java virtual machine. We evaluate and compare performance on a multicore platform running benchmarked versions of large, real-world Java applications. On average across all programs, RT reduces overhead by 49% compared with *strict dependence tracking* (ST)—this paper's term for an existing approach that tracks dependences accurately. In addition to reducing average overhead, RT benefits several high-conflict programs substantially. For programs that incur high coordination costs, RT speeds up execution significantly, achieving on average 84% of the ideal speedup that would be possible if all coordination costs were hidden.

RT's potential is limited by correctness constraints requiring waiting at some program operations; we introduce and evaluate optimizations for overcoming this limitation, but find they have limited benefit. The RT-based recorder outperforms an ST-based recorder by hiding coordination costs, although the improvement is partially offset by recording more events than the ST-based recorder. The RT-based STM's ability to hide coordination costs is

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

CC'16, March 17–18, 2016, Barcelona, Spain
ACM. 978-1-4503-4241-4/16/03...
http://dx.doi.org/10.1145/2892208.2892229

limited by correctness constraints, and its benefit diminishes with more threads. It minimally outperforms an ST-based STM, but for reasons not directly related to hiding coordination costs.

Overall, these results demonstrate the potential for using novel mechanisms to address a key performance bottleneck of parallel runtime support.

2. Background and Motivation

To check or enforce concurrency correctness properties, *parallel runtime support* must track cross-thread dependences (i.e., write–read, write–write, and read–write data dependences involving two threads). In order to track dependences, runtime support must perform synchronized instrumentation at essentially every program memory access. Tracking dependences by using *biased locking* often provides performance advantages, except that lock ownership transfers incur expensive coordination among threads.

2.1 Tracking Cross-Thread Dependences

Runtime support generally needs to *track* cross-thread dependences, which means doing one of the following:

Detecting (identifying) dependences: Runtime support includes data race detectors (e.g., [23, 24]), atomicity checkers (e.g., [6, 25]), and dependence recorders for record & replay (e.g., [11, 36]).

Controlling (enforcing) dependences: Runtime support includes transactional memory (e.g., [28, 62]), memory model enforcement (e.g., [44, 50]), and deterministic execution (e.g., [5, 43]).

For data-race-free (DRF) executions, tracking dependences requires instrumenting only *synchronization* operations, since memory models for shared-memory languages guarantee serializability of synchronization-free regions for DRF executions [1, 2, 9, 38].

However, programs routinely have intentional and unintentional data races [36]. Thus, runtime support must instrument every access that might be involved in a data race.[1] Furthermore, to ensure that dependences are captured soundly, runtime support needs to ensure that an access and its instrumentation execute together atomically, a property we call *instrumentation–access* atomicity.

To preserve instrumentation–access atomicity, runtime support often synchronizes on a lock associated with each object[2] (e.g., represented with an extra word in the object's header). The runtime support's instrumentation acquires the lock for reading and/or writing the object. An implementation typically relies on atomic operations and memory fences, which introduce remote cache misses and serialize in-flight instructions (e.g., [24, 25, 35, 36, 51]).

We note that some approaches have sidestepped these challenges but incur other limitations. For example, record & replay can avoid tracking dependences by relying on replication and speculation, but its performance relies on extra available cores [56]. Some STMs avoid acquiring a lock for every accessed object in a transaction, but sacrifice scalability as a result (e.g., [20]). Some analyses, notably data race detection, need not preserve instrumentation–access atomicity, but still require instrumentation atomicity, which ends up incurring similarly high costs [24, 39].

2.2 Tracking Dependences with Biased Locking

Prior work introduces so-called *biased locking*, in which each object's lock is "biased" toward one thread (or multiple threads in the case of reader locks) [11, 14, 31, 32, 46, 48, 49, 57]. These "owner" thread(s) can reacquire the lock *without* performing an atomic operation or even a store. In order for a thread T to acquire a lock

[1] Even after applying sound (no false negatives) static data race detection as a filter, many accesses cannot be proven to be DRF [17, 23, 36, 58].

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

```
1   if (o.state != WrEx_T) {
2     slowPath(o); /* acquire o.state for write */
3   }
4   o.f = ...; // program store to the object field o.f

5   if (o.state != WrEx_T &&
6       o.state != RdEx_T &&
7       (o.state != RdSh_c || T.rdShCount < c)) {
8     slowPath(o); /* acquire o.state for read */
9   }
10  ... = o.f; // program load to the object field o.f
```

Figure 1. Pseudocode for biased reader–writer locking's instrumentation fast path. T is the executing thread.

```
11 slowPath(o) {
12   state = o.state;
13   // Handle non−conflicting state transitions:
14   if (state == ...) { ...; return; }
15   // Coordination for conflicting transitions:
16   while (state == RdEx*^Int || state == WrEx*^Int
17          || !CAS(&o.state, state, RdEx_T^Int)) { // or WrEx_T^Int
18     state = o.state; // re-read state
19   }
20   coordinate(getOwner(state));
21   o.state = RdEx_T; // or WrEx_T
22 }

23 coordinate(remoteT) {
24   response = sendCoordinationRequest(remoteT);
25   while (!response) {
26     response = status(remoteT);
27   }
28 }
```

Figure 2. Pseudocode for biased reader–writer locking's slow path. T is the executing thread.

owned by other thread(s), T must *coordinate* with the owner(s), so that they do not continue to access the corresponding object racily.

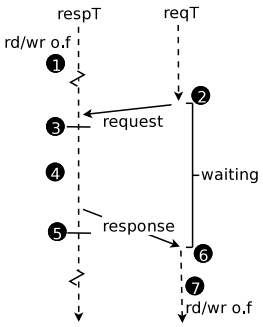
This paper focuses on biased *reader–writer* locks, which support accesses to read-shared data more efficiently than writer locks. We emphasize that our focus is on locks used to implement runtime support—*not* on locks used by programmers in order to enforce mutual exclusion in applications (Section 7).

Without loss of generality, this paper builds on one design of biased reader–writer locks. The rest of this section describes this design, which is most closely based on prior work called *Octet* [11]. Each object's lock can have any of the following states: WrEx_T (write-exclusive for thread T), RdEx_T (read-exclusive for T), or RdSh_c (read-shared for all threads, subject to a counter c that helps detect dependences from the last write soundly [11]). Figure 1 shows the instrumentation that the compiler adds at each program load and store; the instrumentation "acquires" a write or read lock on the accessed object's lock. If a thread already owns the lock for an object, the instrumentation takes the synchronization-free, write-free *fast path*. Otherwise, the instrumentation executes the *slow path*, which changes the state and handles potential cross-thread dependences, as described next.

2.3 Conflicts Require Coordination

When a thread needs to acquire a read or write lock that it does not already own (e.g., at a read by T2 to an object whose lock is in WrEx_{T1} state), it must *coordinate* with the owner thread(s), so they can acknowledge the ownership transfer and cease unsynchronized accesses to the object.

Handling conflicting transitions with coordination. Suppose a thread, called the *requesting thread*, reqT, wants to acquire an ob-



(1) respT accessed an object o previously. (2) reqT wants to access o. It changes o's lock to $RdEx^{Int}_{reqT}$ or $WrEx^{Int}_{reqT}$, 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 lock's state to $WrEx_{reqT}$ or $RdEx_{reqT}$ and proceeds to access o.

Figure 3. Coordination using an explicit request.

ject's lock held in a conflicting state by other thread(s); each of these other thread(s) is a *responding thread*, respT. If the object's lock is in $WrEx_{respT}$ or $RdEx_{respT}$ state, then there is one responding thread, respT. If the object's lock is in RdSh state, 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.

The pseudocode in Figure 2 shows the slow path.[3] First, reqT atomically changes the state of the object's lock to an *intermediate* state, $RdEx^{Int}_{reqT}$ or $WrEx^{Int}_{reqT}$ (depending on whether a read or write lock is needed), with a CAS (line 17).[4] The intermediate state simplifies the protocol by allowing only one thread at a time to perform a conflicting transition on an object's lock. If there is another thread that has already changed the object to an intermediate state, reqT waits for the other thread to finish coordination (lines 17–19).

After reqT successfully changes the object's lock state to $RdEx^{Int}_{reqT}$ or $WrEx^{Int}_{reqT}$, it coordinates with respT (line 20) to ensure that reqT's state change does not interfere with respT's instrumentation–access atomicity. 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—hence preserving instrumentation–access atomicity. Managed language VMs already place safe points at periodic points in compiled code (e.g., method entries and exits and loop back edges) for profiling and timely yielding for parallel garbage collection. *Blocking* operations such as waiting to acquire a program lock or waiting for I/O are also safe points.

If respT is at a *blocking safe point*, reqT makes an *implicit* request to respT (at line 24) by atomically updating respT's status, which respT will see when it leaves the blocking state. The helper method sendCoordinationRequest() returns true if and only if it performs an implicit request. Otherwise, reqT sends an *explicit* request to respT: reqT sends a request to respT by adding a request to respT's *request queue*, and then must wait (lines 25–27) for respT to reach a safe point to respond. While reqT is performing coordination (lines 16–20, including the body of coordinate), it is considered to be at a blocking safe point (mechanism not shown in the pseudocode), to allow other threads to perform implicit requests with reqT acting as a *responding* thread, thus avoiding deadlock. Figure 3 illustrates how coordination works when using an explicit request (this paper's approach modifies only how explicit requests work). Finally, reqT changes the state to $WrEx_{reqT}$ or $RdEx_{reqT}$ (line 21) and proceeds with its access to the field.

Coordination is expensive. As our results show (Section 6), coordination can slow programs substantially, even for programs that perform relatively few conflicting accesses (e.g., 0.1–1% of accesses triggering coordination). The following table reports the average cost of coordination using explicit versus implicit requests, compared with the cost of instrumentation that does not change the lock's state (Section 6.2 describes experimental methodology):

	Same state	Implicit request	Explicit request
CPU cycles	47	360	9,200

Coordination is substantially more expensive than a same-state check because coordination requires several memory accesses, including atomic operations and memory fences. On average, coordination using an explicit request is more than an order of magnitude more costly than using an implicit request, since an explicit request incurs significant latency waiting for roundtrip communication. Our work thus focuses on optimizing coordination that uses explicit requests.

3. Relaxed Dependence Tracking

The last section described prior work's approach for tracking dependences based on biased reader–writer locking. That approach tracks each cross-thread dependence soundly (i.e., does not miss any dependences), by waiting to acquire a read or write lock before proceeding with each access. The rest of this paper refers to that approach as *strict dependence tracking* (ST).

In contrast, this section introduces our novel approach called *relaxed dependence tracking* (RT), which relaxes the instrumentation–access atomicity guarantee provided by ST,[5] allowing threads to continue executing program code without acquiring a dependence tracking lock. The challenge in making RT work lies in preserving both program semantics and runtime-support-specific guarantees.

RT consists of two components: A *relaxed coordination protocol* (Section 3.1) and support for performing *relaxed accesses* that overlap with coordination (Section 3.2).

3.1 The Relaxed Coordination Protocol

In RT, a requesting thread does *not* wait for responses after sending 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 simultaneously. To support this functionality, the relaxed coordination protocol differs from the strict coordination protocol 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.

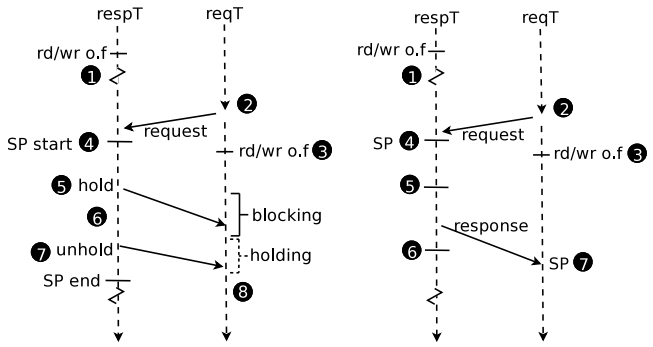
Figure 4 shows how the relaxed coordination protocol works. reqT sends an explicit request to respT and continues execution. When respT reaches a safe point, it responds to reqT either explicitly or implicitly. If reqT is blocked, respT responds implicitly, as shown in Figure 4(a), by first putting reqT into a "blocked and held" state (so that reqT does not leave the blocking state unless it is "unheld") and then changing the object's lock's state. Finally, the responding thread removes its hold on the requesting thread. Otherwise (reqT is not blocked), respT responds explicitly, as Figure 4(b) shows, by adding a response to reqT's queue. Once reqT reaches a safe point, it changes the object's lock's state.

Although Figure 4 shows a single requesting thread sending requests to respT, multiple requesting threads can send requests to

[3] The pseudocode omits memory fences required by the implementation.

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

[5] Although our design of RT targets coordination latency introduced by biased reader–writer locking, it should be possible to adapt RT to other dependence-tracking mechanisms.



(a) **Implicit response:** (1) respT accessed o at some prior time. (2) reqT wants to access o. It changes o's lock to $\text{RdEx}^{\text{Int}}_{\text{reqT}}$ or $\text{WrEx}^{\text{Int}}_{\text{reqT}}$. (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 lock's state to $\text{WrEx}_{\text{reqT}}$ or $\text{RdEx}_{\text{reqT}}$. (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 lock to $\text{RdEx}^{\text{Int}}_{\text{reqT}}$ or $\text{WrEx}^{\text{Int}}_{\text{reqT}}$. (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 lock's state to $\text{WrEx}_{\text{reqT}}$ or $\text{RdEx}_{\text{reqT}}$.

Figure 4. The *relaxed* coordination protocol (for explicit requests only).

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

Figures 5 and 6 show the pseudocode for RT's load and store instrumentation. RT uses the same fast path as ST (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, ST initiates coordination by changing the state of the object to $\text{WrEx}^{\text{Int}}_{\text{T}}$ (line 14 in Figure 5) or $\text{RdEx}^{\text{Int}}_{\text{T}}$ (line 17 in Figure 6) and sending a request to the responding thread (line 15 in Figure 5 and line 18 in Figure 6). After sending the coordination request, T continues execution immediately, subject to constraints about accessing objects that are not yet locked in the needed state.

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 blocking safe point. Before T receives a response, the conflicting object o stays in the $\text{WrEx}^{\text{Int}}_{\text{T}}$ or $\text{RdEx}^{\text{Int}}_{\text{T}}$ state, since both T and other threads might perform accesses to it. If the access is a store, and o is in $\text{RdEx}^{\text{Int}}_{\text{T}}$ state, RT upgrades the state to $\text{WrEx}^{\text{Int}}_{\text{T}}$ (line 10 in Figure 5), in order to track relaxed stores, introduced shortly. If o is locked in $\text{WrEx}^{\text{Int}}_{*}$ but not $\text{WrEx}^{\text{Int}}_{\text{T}}$ (i.e., other threads performing relaxed stores to o), 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 $\text{WrEx}^{\text{Int}}_{*}$ or $\text{RdEx}^{\text{Int}}_{*}$, the access can avoid performing coordination.

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.

```
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 == ...) { ...; *addr = newValue; return; }
10   if (state == RdEx^Int_T) { /* upgrading trans. to WrEx^Int_T */ }
11   boolean relaxed = true;
12   while (state != WrEx^Int_T) {
13     if (state != WrEx^Int_* &&
14         CAS(&o.state, state, WrEx^Int_T)) {
15       relaxed = !sendCoordinationRequest(getOwner(state));
16       break;
17     }
18     state = o.state; // re−read state
19   }
20   if (relaxed) {
21     storeBufferSet(addr, newValue); // defer the store
22   } else {
23     o.state = WrEx_T;
24     *addr = newValue;
25   }
26 }
```

Figure 5. 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^Int_T && storeBufferContains(addr))
13     return storeBufferGet(addr);
14   boolean relaxed = true;
15   while (state != WrEx^Int_* && state != RdEx^Int_*) {
16     // Coordination for conflicting transition :
17     if (CAS(&o.state, state, RdEx^Int_T)) {
18       relaxed = !sendCoordinateRequest(getOwner(state));
19       break;
20     }
21     state = o.state;  // re-read state
22   }
23   value = *addr;
24   if (relaxed)
25     logLoadedValue(addr, value);
26   else
27     o.state = RdEx_T;
28   return value;
29 }
```

Figure 6. The fast and slow paths of RT's instrumentation at loads.

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.

3.2 Handling Relaxed Accesses

A thread T performs *relaxed accesses*[6] to objects whose locks are not (yet) in the needed state. RT defers a *relaxed store* until it receives coordination response(s) for the object's lock. 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 lock. 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 21 in Figure 5). 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 lock's state to $\mathsf{WrEx}_{\mathsf{T}}$), it performs all deferred stores to o using the store buffer.

For simplicity, our current design limits relaxed stores by T to objects locked in $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T}}$ state. (We have found that supporting relaxed stores to other lock 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 [1, 2, 9, 38], 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 program synchronization *release* operations (e.g., lock release, thread fork, and Java volatile or C++ atomic writes).

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 12 in Figure 6). (T only needs to check its store buffer if o's lock is in $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T}}$ state.) If so, T uses the store buffer's value (line 13 in Figure 6) instead of loading from memory. Otherwise, T performs the load directly from memory (line 23 in Figure 6). A relaxed load thus does *not* affect program semantics: the execution still conforms to the memory model (since performing the load would be permitted in the original program). However, a relaxed load could certainly impact the correctness of runtime support that needs to detect or control cross-thread dependences. In particular, 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.

RT thus handles each relaxed load by *logging the loaded value* in a *runtime-support-specific way* (line 25 in Figure 6). The intuition is that logging the value enables runtime support to handle all values resulting from potentially untracked cross-thread dependences. For example, our RT-based dependence recorder logs the value in order to assist replay (Section 4), and our RT-based STM logs the value in order to validate it later (Section 5).

3.3 Optimizations at Synchronization Release Operations

As presented so far, a thread must wait at each program synchronization release operation for every outstanding deferred 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 release operations. 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 deferred 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 deferred relaxed stores that executed before the lock release (according to bookkeeping).

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 deferred stores. A straightforward way to provide this restriction is to disallow all accesses by other threads to objects locked in $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T}}$ 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 locked in $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T1}}$ state.

4. Recording Dependences

This section shows how to use RT to optimize *multithreaded record & replay*, which enables deterministically replaying a recorded execution. Record & replay enables both offline debugging [36, 56] and system features such as replication-based fault tolerance [12, 37]. Recording dependences efficiently is the key challenge of multithreaded record & replay. However, recording dependences is expensive due to the high cost of tracking dependences between all potentially shared memory accesses [35, 36].

4.1 ST-Based Dependence Recorder

Prior work builds a dependence recorder, which we call the *ST-based recorder*, on top of biased reader–writer locks that use strict dependence tracking [11]. It records all happens-before edges [34] at transitions between WrEx, RdEx, and RdSh states. It records each happens-before edge by recording its *source* and *sink*, each of which is recorded as a *dynamic program location* (DPL), which consists of a static program location (e.g., method and line number) and a per-thread counter incremented at every method entry, method exit, and loop back edge. Another execution can replay these happens-before edges deterministically by making the sink wait for its corresponding source to be reached.

4.2 RT-Based Dependence Recorder

Our *RT-based dependence recorder* extends the ST-based recorder by using relaxed, instead of strict, dependence tracking. The RT-based 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 dependences accurately and thus guarantee deterministic replay?

We refer back to Figure 4 on page 4 for examples of happens-before edges recorded by the RT-based 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

[6] This paper's *relaxed accesses* should not be confused with memory_order_relaxed operations on atomic variables in C/C++ [9].

#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 deferred stores to objects locked in $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{reqT}}$ state, the RT-based recorder uses the following strategy: reqT records an event for each deferred 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 deferred 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 whose lock 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 4(a) and Figure 4(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 3.2), 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.

Value determinism. Our RT-based recorder provides *value determinism*, i.e., each load reads the same value during replay as during record. If a load was relaxed during record, it will load from the log during replay. Notably, this value will match the value of the variable in memory (i.e., the value that *would* normally have been loaded from memory) unless there is a data race. Other efficient record & replay techniques, such as DoublePlay [56] and Respec [37], provide *output determinism*, which is weaker than value determinism but is still useful for both offline replay (e.g., replaying buggy executions) and online replay (e.g., replicating a multithreaded process).

5. Software Transactional Memory

This section describes how we extend an existing software transactional memory (STM) system to use RT. In essence, our *RT-based 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.

5.1 ST-Based STM

Prior work introduces an STM that uses biased reader–writer locks that employ strict dependence tracking [62]. We call this STM the *ST-based STM*. The ST-based 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.

5.2 RT-Based STM

Extending the ST-based 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 7

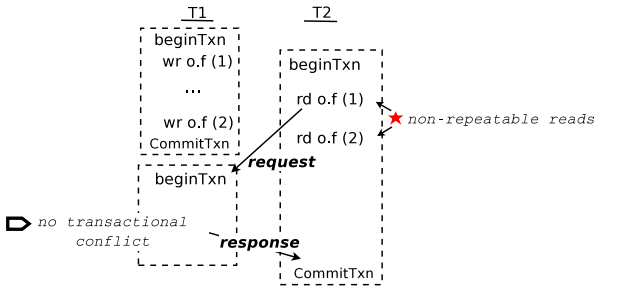


Figure 7. Allowing unhandled relaxed accesses in transactions would lead to serializability violations. The values in parentheses after each executed store and load are the values written and read, respectively.

shows an example of a problematic execution. Thread T2 performs two relaxed loads from o.f in a transaction, since o's lock is in $\mathsf{WrEx}_{\mathsf{T1}}$ state. 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 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 *RT-based STM* addresses these issues as follows. In the RT-based STM, each relaxed, transactional load logs its loaded value, and *validates* the value later. (In Figure 7, T2's transaction would fail the read validation before commit and abort.) Each relaxed, transactional store is deferred, to provide opacity of intermediate updates before the transaction commits. Before a transaction commits, it waits for coordination responses so it can validate all relaxed loads and perform all deferred stores.

Relaxed loads. A requesting thread reqT performs a relaxed load when it reads from an object o locked in the $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{reqT}}$ or $\mathsf{RdEx}^{\mathsf{Int}}_{\mathsf{reqT}}$ state. reqT first checks its store buffer for the value (only if the object's state is $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{reqT}}$). If not found, 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. If validation fails, reqT must abort its current transaction. (If respT responds implicitly, 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 [29]. 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, 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 deferred stores. Unlike our general RT design, our RT-based STM

does *not* support loads by T to objects locked in intermediate states *other than* $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T}}$ and $\mathsf{RdEx}^{\mathsf{Int}}_{\mathsf{T}}$; supporting loads from other states would require a mechanism for eventually changing the lock's state to $\mathsf{WrEx}_{\mathsf{T}}$, $\mathsf{RdEx}_{\mathsf{T}}$, or RdSh in order to validate reads before commit.

Guaranteeing progress. The *ST-based* STM guarantees progress by detecting all conflicts eagerly and then aborting the younger transaction [54, 62]. However, the *RT-based* STM cannot guarantee progress, since any transaction that fails read validation must abort. Other mixed-mode STMs have similarly lacked progress guarantees [30, 47]. Standard techniques such as exponential backoff can help to alleviate livelock. For the RT-based STM, a simple (unimplemented) solution exists: if a transaction repeatedly fails read validation, it falls back to use *strict* dependence tracking, guaranteeing it will commit (at least once it becomes oldest).

Correctness. At a high level, the RT-based 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 ST, eager conflict detection and resolution guarantee that conflicting accesses between the committing transaction's accesses and commit time will be detected and resolved. (The RT-based 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 [29]. In managed languages such as Java, zombies are not a serious problem because memory and type safety are preserved [19, 40]. Targeting a native language such as C++ would require additional support to provide *sandboxing* of zombie transactions [19]. Another issue is that zombie transactions can get stuck in infinite loops that are impossible in any serializable execution. The RT-based STM cannot experience this issue because it validates relaxed loads within a bounded amount of time.

Comparison with prior work. Most STMs employ either entirely lazy or entirely eager concurrency control (e.g., [20, 22, 54, 62]). Some STMs combine lazy and eager mechanisms, by using eager concurrency control for *writes* and lazy validation for *reads* [30, 47]. The RT-based STM combines eager and lazy concurrency control in a novel way, using eager and lazy concurrency control for non-conflicting and conflicting accesses, respectively.

6. Evaluation

This section first evaluates the performance and run-time characteristics of relaxed dependence tracking (RT) without runtime support, compared with strict dependence tracking (ST). It then evaluates the two case studies of runtime support built upon RT, compared with ST-based approaches.

6.1 Implementation

We have implemented RT and the RT-based recorder and STM in Jikes RVM, a Java virtual machine (JVM) [3, 4] that performs competitively with commercial JVMs [7]. We have made our implementations publicly available on the Jikes RVM Research Archive.

Our RT implementation builds on the publicly available ST implementation called *Octet* [11]. Our RT-based recorder builds on the publicly available ST-based recorder [10, 11]. Our RT-based STM builds on the publicly available ST-based STM called *LarkTM* [62]. Our implementations reuse features of the ST-based implementations as much as possible.

6.2 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) [8], excluding single-threaded programs and programs that Jikes RVM cannot execute
- *Business logic benchmarks:* fixed-workload versions of SPECjbb2000 and SPECjbb2005[7]
- *Medium-sized benchmarks that stress various multithreaded execution scenarios:* the Java Grande benchmarks (excluding microbenchmarks) [53]
- *Transactional benchmarks:* the STAMP benchmarks [16], ported to Java providing six working programs [21, 33, 62]

Experimental setup. For each implementation, we build a high-performance configuration of Jikes RVM. Each performance result is the median of 25 trials. We also show the mean, as the center of 95% confidence intervals.

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

6.3 Evaluating Relaxed Dependence Tracking

This section evaluates RT and compares it with ST, by executing instrumentation that tracks dependences but provides no runtime support on top of it.

Measuring the problem. We first measure the cost of ST, as well as the maximum benefit that can be obtained from optimizations. Figure 8 shows runtime overhead of three configurations over an unmodified JVM. Each configuration instruments accesses to track dependences using biased reader–writer locks. The first configuration, *ST*, uses strict tracking (as described in Section 2.3) and adds 139% overhead on average. This overhead varies considerably across the evaluated programs; the overhead for each program is closely linked to the fraction of accesses that trigger coordination using explicit requests (as shown later in this section), which is the main cost of tracking dependences.

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. This configuration attempts to estimate an upper bound on the performance that RT might be able to provide. On average, *Ideal* adds 82% 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 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. Figure 8's last configuration, *RT (no stall at unlock)*, uses relaxed dependence tracking with the second optimization from Section 3.3. Its overhead over baseline execution is 90%, a significant reduction (49% relative to baseline execution time) from ST's 139% overhead. Furthermore, RT achieves much of the maximum possible benefit, approaching *Ideal*'s 82% overhead.

We note that for sparse, RT significantly outperforms the *Ideal* configuration. This counterintuitive result is due to the fact that the *Ideal* configuration estimates the cost of coordination without latency, but it does not account for potential other improvements that RT might provide. As we show later in this section, RT can reduce the number of conflicting transitions (compared with ST);

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

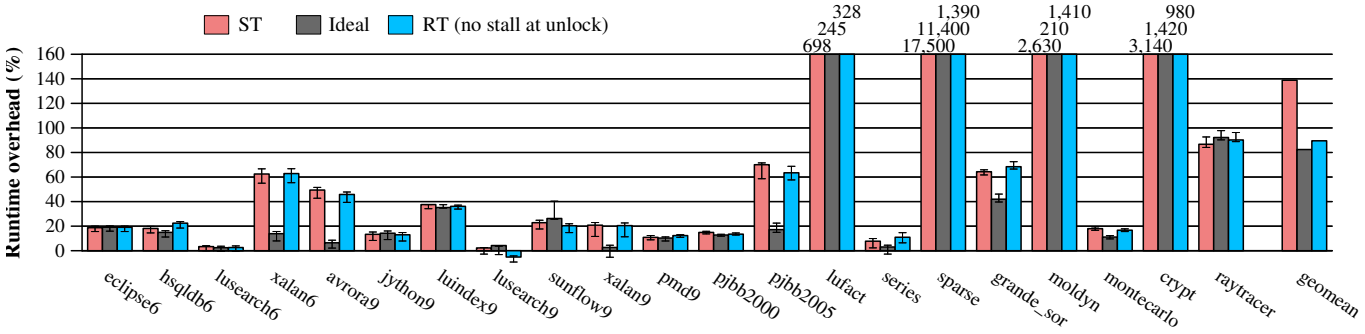


Figure 8. Run-time overhead added to an unmodified JVM by capturing dependences using (1) ST, compared with (2) an ideal, unsound configuration that eliminates coordination latency and (3) RT.

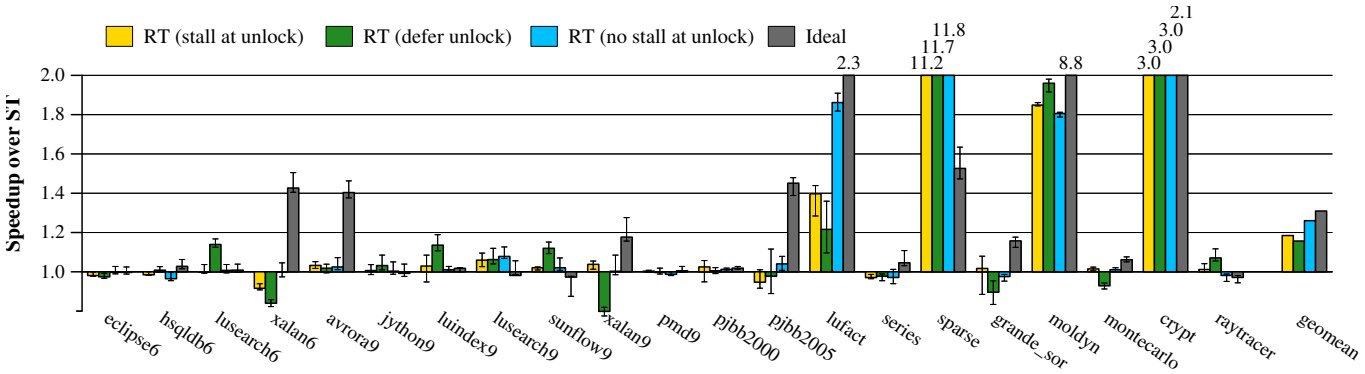


Figure 9. Speedup of RT relative to ST. *Ideal* is an unsound configuration that provides an upper bound on RT's performance.

RT reduces sparse's conflicting transitions substantially, leading to significantly lower overhead than for *Ideal*.

Figure 9 is a *speedup* graph (higher is better) that shows the same configurations as Figure 8 plus two additional RT configurations. The RT configurations, which are all sound, are as follows:

RT (stall at unlock): The default design from Section 3. Threads wait at synchronization release operations for all outstanding relaxed stores.

RT (defer unlock): The first optimization described in Section 3.3. At a lock release, a thread defers the lock release if there are outstanding relaxed stores.

RT (no stall at unlock): The second optimization described in Section 3.3 (and also shown in Figure 8). 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 locked in $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T}}$ state.

The three RT configurations each provide an average speedup of 1.16–1.26X over ST. These speedups are not far from the average speedups achieved by *Ideal* (1.31X), suggesting that RT is getting most of the maximum possible benefit from hiding the cost of coordination due to explicit requests.

While the RT configurations outperform *ST* and get close to *Ideal*'s performance on average, RT does not help much with the gap between *ST* and *Ideal* in several cases (hsqldb6, xalan6, avrora9, xalan9, grand_sor, and montecarlo). As we show later, RT changes the balance of explicit versus implicit coordination requests (relative to ST), by causing threads to spend more time executing code instead of blocking at safe points. This change cancels out RT's potential performance benefits in several cases, 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 ST, and keeping track of relaxed events and accesses requires performing additional work.

The *RT (defer unlock)* configuration does not improve performance on average (nor significantly for any individual program) compared with the *RT (stall at unlock)*. 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 operation waiting for coordination responses. These factors are enough to outweigh any potential benefit provided by deferring unlocks.

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

Run-time characteristics. Next we focus on understanding factors contributing to the performance difference between RT and ST. Table 1 reports runtime statistics for tracking dependences with RT and ST. The table uses the RT configuration *RT (no stall at unlock)*.

For each type of coordination, *Lock state transitions* counts how many accesses execute instrumentation that requires either no lock state change (*Same state*) or a *Conflicting* transition that triggers coordination. An interesting point is that RT sometimes reduces *how many* conflicting transitions occur, relative to ST. This phenomenon occurs because of cases in which an object is heavily contended, and two or more threads repeatedly transfer its ownership in quick succession. When using ST, 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

	Strict dependence tracking				Relaxed dependence tracking							
	Lock state transitions		Coord. requests		Lock 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}	1.4×10^{5} (0.0011%)	1.6×10^{4}	2.9×10^{5}	1.2×10^{10}	1.4×10^{5} (0.0011%)	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}	9.0×10^{5} (0.14%)	3.5×10^{4}	3.8×10^{6}	6.2×10^{8}	9.0×10^{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}	4.4×10^{3} (0.00018%)	2.3×10^{3}	4.5×10^{3}	2.4×10^{9}	4.4×10^{3} (0.00018%)	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}	1.9×10^{7} (0.17%)	1.3×10^{7}	5.9×10^{6}	1.1×10^{10}	1.9×10^{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}	5.9×10^{6} (0.097%)	4.1×10^{6}	1.8×10^{7}	6.1×10^{9}	5.7×10^{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}	6.6×10^{1} (0.0000013%)	1.8×10^{1}	1.5×10^{0}	5.1×10^{9}	6.2×10^{1} (0.0000012%)	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}	3.7×10^{2} (0.00011%)	1.5×10^{1}	3.3×10^{2}	3.5×10^{8}	3.7×10^{2} (0.00011%)	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}	2.9×10^{3} (0.00012%)	4.6×10^{3}	4.4×10^{3}	2.4×10^{9}	2.9×10^{3} (0.00012%)	5.0×10^{3}	3.3×10^{3}	1.3×10^{3}	3.7×10^{3}	6.4×10^{3}	4.1×10^{2}
sunflow9	1.7×10^{10}	1.4×10^{4} (0.000078%)	1.5×10^{4}	7.6×10^{3}	1.7×10^{10}	9.3×10^{3} (0.000054%)	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}	1.8×10^{7} (0.18%)	9.7×10^{6}	8.7×10^{6}	1.0×10^{10}	2.0×10^{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}
pmd9	5.7×10^{8}	4.4×10^{4} (0.0077%)	3.1×10^{4}	5.3×10^{4}	5.7×10^{8}	4.3×10^{4} (0.0075%)	2.7×10^{4}	4.9×10^{4}	2.0×10^{4}	6.9×10^{3}	2.5×10^{4}	3.4×10^{4}
pjbb2000	1.7×10^{9}	9.5×10^{5} (0.055%)	6.2×10^{4}	9.0×10^{5}	1.7×10^{9}	9.5×10^{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}	4.6×10^{7} (0.69%)	3.2×10^{7}	5.7×10^{7}	6.5×10^{9}	4.1×10^{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}
lufact	8.0×10^{9}	6.0×10^{5} (0.0075%)	5.3×10^{6}	1.4×10^{6}	8.4×10^{9}	5.2×10^{5} (0.0061%)	8.5×10^{5}	8.4×10^{5}	3.3×10^{5}	5.3×10^{5}	1.2×10^{7}	3.1×10^{5}
series	4.0×10^{6}	2.0×10^{6} (33%)	2.0×10^{6}	3.0×10^{4}	4.0×10^{6}	2.0×10^{6} (33%)	1.4×10^{6}	6.2×10^{5}	1.2×10^{6}	1.4×10^{5}	1.2×10^{2}	1.4×10^{6}
sparse	6.7×10^{9}	2.4×10^{8} (3.4%)	3.8×10^{8}	8.3×10^{7}	6.0×10^{9}	4.5×10^{7} (0.74%)	3.1×10^{7}	1.8×10^{7}	2.8×10^{7}	3.3×10^{6}	5.0×10^{8}	4.9×10^{8}
grande_sor	3.7×10^{9}	3.9×10^{4} (0.0011%)	4.2×10^{5}	1.2×10^{5}	3.6×10^{9}	3.9×10^{4} (0.0011%)	3.7×10^{5}	1.3×10^{5}	4.2×10^{4}	3.3×10^{5}	8.8×10^{4}	2.8×10^{4}
moldyn	4.0×10^{10}	9.7×10^{7} (0.25%)	1.7×10^{8}	8.0×10^{7}	3.3×10^{10}	8.5×10^{7} (0.26%)	7.8×10^{7}	5.3×10^{7}	4.9×10^{7}	2.9×10^{7}	6.6×10^{7}	5.3×10^{7}
montecarlo	2.4×10^{9}	5.2×10^{5} (0.022%)	3.2×10^{5}	2.0×10^{5}	2.4×10^{9}	4.3×10^{5} (0.018%)	2.7×10^{5}	1.5×10^{5}	2.5×10^{5}	2.2×10^{4}	2.2×10^{5}	3.1×10^{5}
crypt	5.2×10^{8}	3.9×10^{7} (7.0%)	3.9×10^{7}	3.2×10^{5}	5.2×10^{8}	7.4×10^{6} (1.4%)	4.8×10^{6}	2.6×10^{6}	4.8×10^{6}	1.1×10^{4}	2.2×10^{3}	3.7×10^{7}
raytracer	3.3×10^{10}	1.1×10^{4} (0.000032%)	7.1×10^{3}	5.2×10^{3}	3.3×10^{10}	1.1×10^{4} (0.000032%)	5.6×10^{3}	6.4×10^{3}	4.4×10^{3}	1.2×10^{3}	1.6×10^{5}	1.6×10^{3}

Table 1. Run-time characteristics of strict and relaxed dependence tracking.

using RT—particularly when executing past release operations as permitted by the *RT no stall at unlock* configuration—a thread is more likely to perform consecutive *relaxed* accesses to a contended object, leading to fewer conflicting transitions, compared with ST. This effect is directly responsible for RT outperforming the *Ideal* configuration for sparse.

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 immediately follow coordination requests, others are repeat accesses to the same memory 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.

6.4 Performance of Runtime Support

This section evaluates whether RT can benefit runtime support that detects cross-thread dependences (dependence recorder) and controls cross-thread dependences (STM).

Dependence recorder. Figure 10 shows the performance of the ST- and RT-based recorders. Not surprisingly, the performance story for the recorders is similar to the story for tracking dependences alone. On average, the RT recorder is 1.23X faster than the ST-based recorder, and four benchmarks show large improvements.

We note that although RT can hide coordination latency, the RT-based 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 RT-based recorder thus often logs *more* than the ST-based recorder. Notably, the RT-based recorder's log size is about 2X the ST-based recorder's for lusearch6, xalan6, and xalan9; and about 6X for grande_sor. For all other programs, the RT-based recorder logs less than 50% more than the ST-based recorder.

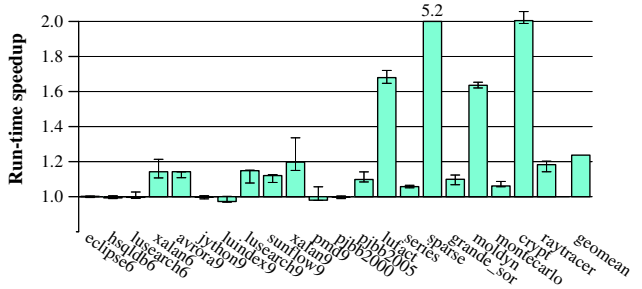


Figure 10. Run-time speedup of the RT-based dependence recorder over the ST-based dependence recorder.

Software transactional memory. Next we compare the ST- and RT-based STMs using the transactional STAMP benchmarks. Figure 11 shows the execution time of the ST- and RT-based STMs. We first note that both STMs typically scale poorly after 8 threads; prior work has also found that STAMP has limited scalability [63]. Furthermore, our platform has 8 cores per socket, leading to greater inter-thread communication for >8 threads.

For genome and vacation, RT reduces overhead for 2–8 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 ST for 8 to 32 threads. For these programs, the RT-based STM achieves a significantly lower rate of aborting transactions than the ST-based STM. The RT-based STM validates relaxed loads at object field and array element granularity, as opposed to the ST-based 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

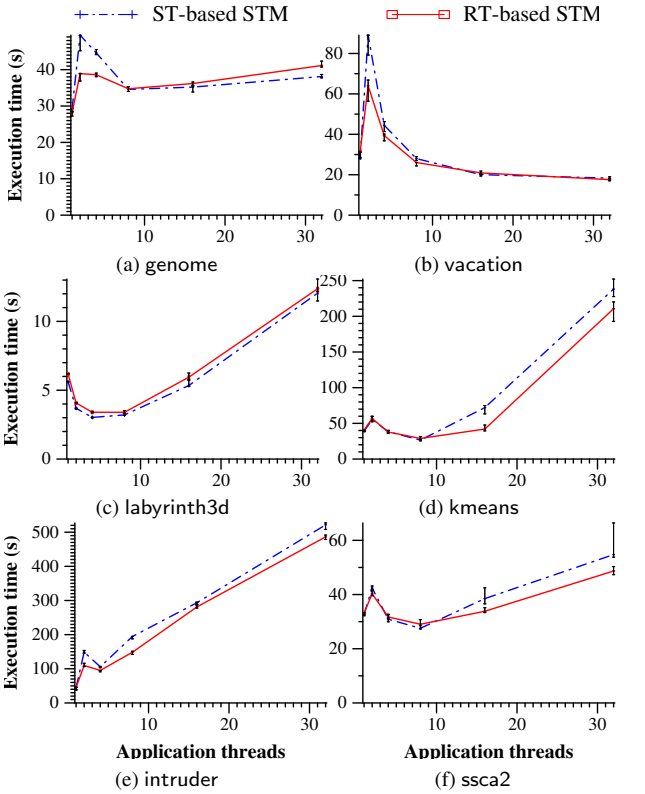


Figure 11. Performance of the ST- and RT-based STMs.

many transactions are short, and the RT-based STM must wait at transaction end for all outstanding relaxed accesses (Section 5.2).

In summary, RT reduces the cost of tracking dependences significantly, particularly for programs with high coordination costs, recovering much of the maximum possible performance achievable by eliminating coordination latency entirely. RT's 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 ST). Although the RT-based 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 compares relaxed dependence tracking to other approaches not already covered in Section 2 and the end of Section 5.2.

Targeting synchronization costs. Prior work has targeted the costs of using biased locking to provide instrumentation–access atomicity for dependence tracking. Cao et al. hybridize biased and unbiased locking [15]. Von Praun and Gross use a model that switches to an unbiased state for shared objects [57], but unbiased states require atomic operations, leading to high overhead [11].

Neamtiu and Hicks introduce "relaxed synchronization" to allow threads to keep executing while waiting to join a global synchronization barrier (for dynamic software update) [42]. Their work does not track dependences. We note that RT not only relaxes synchronization, but it relaxes dependence tracking guarantees.

Biased program locks. This paper focuses on one context for biased locking: providing instrumentation–access atomicity for capturing cross-thread dependences. Other work has used biased locking for program locks [32, 46, 55]. Biased program locks typically mitigate coordination costs by switching a conflicting lock to an unbiased state after the first conflict.

Hardware support. Intel's recent Transactional Extensions (TSX) provide limited, best-effort hardware support for speculative atomicity [61]. TSX could potentially help solve the same problem as software-based reader–writer locks. However, recent work suggests that TSX struggles to outperform even atomic-operation-based synchronization if transactions are short [39, 45, 61]. In contrast, biased reader–writer locks avoid atomic operations. An empirical comparison is beyond this paper's scope.

8. Conclusion

Runtime support based on biased reader–writer locking can achieve significantly lower overhead than traditional locks, but conflicting accesses require threads to coordinate with each other. Relaxed dependence tracking (RT) seeks to relax the requirement of tracking all dependences soundly, by allowing coordination latency to overlap with useful program work. RT's novel approach provides modest benefit overall since (1) many programs incur low coordination costs to begin with; (2) RT cannot hide all costs, particularly the cost of remote cache misses; and (3) correctness requirements limit relaxation at program synchronization release operations and transaction commits. This work contributes to the design of parallel runtime support, by showing that opportunities exist to exploit the flexibility of program semantics and the mechanics of sound runtime support.

Acknowledgments

We thank our shepherd, Matthew Flatt, and the anonymous reviewers for their insightful comments and suggestions; and Man Cao and Aritra Sengupta for helpful discussions.

References

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

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

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

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

[5] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *ASPLOS*, pages 53–64, 2010.

[6] S. Biswas, J. Huang, A. Sengupta, and M. D. Bond. DoubleChecker: Efficient Sound and Precise Atomicity Checking. In *PLDI*, pages 28–39, 2014.

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

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

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

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

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

[12] T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault Tolerance. In *SOSP*, pages 1–11, 1995.

[13] N. G. Bronson, C. Kozyrakis, and K. Olukotun. Feedback-Directed Barrier Optimization in a Strongly Isolated STM. In *POPL*, pages 213–225, 2009.

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

[15] M. Cao, M. Zhang, A. Sengupta, and M. D. Bond. Drinking from Both Glasses: Combining Pessimistic and Optimistic Tracking of Cross-Thread Dependences. In *PPoPP*, 2016. To appear.

[16] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC*, 2008.

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

[18] J.-D. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *SPDT*, pages 48–59, 1998.

[19] L. Dalessandro and M. L. Scott. Sandboxing Transactional Memory. In *PACT*, pages 171–180, 2012.

[20] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *PPoPP*, pages 67–78, 2010.

[21] B. Demsky and A. Dash. Evaluating Contention Management Using Discrete Event Simulation. In *TRANSACT*, 2010.

[22] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching Transactional Memory. In *PLDI*, pages 155–165, 2009.

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

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

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

[26] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *OOPSLA*, pages 57–76, 2007.

[27] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. JaRec: A Portable Record/Replay Environment for Multi-threaded Java Applications. *SPE*, 34(6):523–547, 2004.

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

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

[30] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *PLDI*, pages 14–25, 2006.

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

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

[33] G. Korland, N. Shavit, and P. Felber. Deuce: Noninvasive Software Transactional Memory in Java. *Transactions on HiPEAC*, 5(2), 2010.

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

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

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

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

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

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

[40] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *SPAA*, pages 314–325, 2008.

[41] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Single Global Lock Semantics in a Weakly Atomic STM. In *TRANSACT*, 2008.

[42] I. Neamtiu and M. Hicks. Safe and Timely Updates to Multi-threaded Programs. In *PLDI*, pages 13–24, 2009.

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

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

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

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

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

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

[49] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-Grain Access Control for Distributed Shared Memory. In *ASPLOS*, pages 297–306, 1994.

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

[51] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing Isolation and Ordering in STM. In *PLDI*, pages 78–88, 2007.

[52] J. M. Silva, J. Simão, and L. Veiga. Ditto – Deterministic Execution Replayability-as-a-Service for Java VM on Multiprocessors. In *Middleware*, pages 405–424, 2013.

[53] L. A. Smith, J. M. Bull, and J. Obdrzálek. A Parallel Java Grande Benchmark Suite. In *SC*, pages 8–8, 2001.

[54] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A Comprehensive Strategy for Contention Management in Software Transactional Memory. In *PPoPP*, pages 141–150, 2009.

[55] N. Vasudevan, K. S. Namjoshi, and S. A. Edwards. Simple and Fast Biased Locks. In *PACT*, pages 65–74, 2010.

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

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

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

[59] P. Wu, M. M. Michael, C. von Praun, T. Nakaike, R. Bordawekar, H. W. Cain, C. Cascaval, S. Chatterjee, S. Chiras, R. Hou, M. Mergen, X. Shen, M. F. Spear, H. Y. Wang, and K. Wang. Compiler and Runtime Techniques for Software Transactional Memory Optimization. *Concurrency and Computation: Practice and Experience*, 21(1):7–23, 2009.

[60] Z. Yang, M. Yang, L. Xu, H. Chen, and B. Zang. ORDER: Object Centric Deterministic Replay for Java. In *USENIX*, pages 30–30, 2011.

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

[62] M. Zhang, J. Huang, M. Cao, and M. D. Bond. Low-Overhead Software Transactional Memory with Progress Guarantees and Strong Semantics. In *PPoPP*, pages 97–108, 2015.

[63] F. Zyulkyarov, S. Stipic, T. Harris, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. Discovering and Understanding Performance Bottlenecks in Transactional Applications. In *PACT*, pages 285–294, 2010.