Optimizing Parallel Runtime Support with Asynchronous Coordination

Ohio State CSE technical report #OSU-CISRC-11/15-TR23, Nov 2015

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

Abstract

Although shared memory provides a simple and efficient model, achieving both correctness and scalability is hard. This paper focuses on providing *runtime support*—such as multithreaded record & replay and software transactional memory—that enhances the correctness and scalability of parallel programs. Existing runtime support captures cross-thread dependences efficiently by using synchronization that avoids atomic operations at non-conflicting memory accesses, but requires threads to *coordinate* whenever conflicts do occur. Coordination slows execution significantly, even for programs with modest amounts of inter-thread communication.

This paper investigates the potential for runtime support that captures cross-thread dependences using coordination that is *asynchronous*, enabling threads to continue execution without waiting for coordination to complete. The key challenge in designing our *asynchronous coordination* (AC) approach is preserving both the language's semantics and the runtime support's guarantees. We design, implement, and evaluate AC and two types of AC-based runtime support. Our evaluation shows that AC often provides significant performance benefits for programs that can benefit from it (programs that incur high coordination costs). AC cannot hide all of coordination's latency, due to limitations related to correctness constraints and performance side effects from AC's mechanisms. Overall, this work demonstrates the potential for identifying and exploiting asynchrony in high-latency synchronization mechanisms.

1. Introduction

Software must become more parallel in order to fully utilize hardware that provides more, instead of faster, cores in successive generations. However, developing shared-memory parallel programs that are correct and scalable is notoriously difficult. To this end, researchers and practitioners have developed various kinds of *runtime support* for making parallel software correct and scalable. For example, *multithreaded record & replay* enables offline debugging and online replication [17, 25, 26, 35, 49, 52, 56]. *Transactional memory* enforces atomicity without the correctness and performance challenges associated with using fine-grained locking [12, 20, 27, 29, 30, 32, 38, 39, 44, 48, 55, 58]. However, existing runtime support is *impractical*: it relies on unrealistic custom hardware, adds high overhead, or incurs other serious limitations.

This paper focuses on runtime support for commodity systems, often called "software only." Runtime support generally needs to *capture* (track or control) cross-thread data dependences soundly, which entails using *synchronized* instrumentation to ensure that instrumentation and program access execute together atomically (Section 2.1). Several existing approaches attempt to minimize the cost of this synchronized instrumentation by using mechanisms based on *biased reader–writer locking*, in which thread(s) release a lock only when another thread needs the lock for conflicting accesses, avoiding the costs of reacquiring locks for non-conflicting accesses [10, 13, 30, 31, 43, 45, 46, 53] (Section 2.2). Despite the advantages of runtime support based on biased reader–writer locking, a serious drawback is that transferring lock ownership requires threads to *coordinate* with each other—a high-latency operation that slows executions substantially that perform even a modest amount of shared-memory communication (Section 2.3).

This paper explores the potential of ameliorating coordination costs by *overlapping coordination with program execution*—while preserving both the runtime support's guarantees and the language's semantics. We introduce *asynchronous coordination* (AC), which enables a thread to continue executing as soon as it initiates coordination and to receive coordination responses later. AC consists of two components: an *AC protocol* and *asynchronous memory accesses (loads and stores)*. We show that in order to overlap asynchronous loads and stores with the AC protocol correctly, asynchronous stores should be deferred, and asynchronous loads should be logged and handled in a runtime-support-specific way.

To demonstrate that AC can be a building block for various runtime support, we design and implement two kinds of runtime support: (1) an *asynchronous dependence recorder* that builds on an existing (synchronous) dependence recorder that uses biased reader–writer locks to detect and record cross-thread dependences [9, 10]; and (2) an *asynchronous software transactional memory (STM)* that builds on an existing (synchronous) STM that uses biased reader–writer locking for concurrency control [58].

We have implemented AC and the asynchronous 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, AC reduces overhead by 49% compared with synchronous coordination. In addition to the average overhead being reduced, AC benefits several high-conflict programs substantially. For programs that incur high coordination costs, AC speeds up execution significantly, achieving 72% on average of the ideal speedup possible from hiding all coordination costs. AC'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 asynchronous recorder outperforms the synchronous recorder by hiding coordination costs, although the improvement is partially offset by recording more events than the synchronous recorder in some cases. The asynchronous STM's ability to hide coordination costs is limited by correctness constraints; it minimally outperforms the synchronous 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 Related Work

In order to capture inter-thread behaviors soundly, runtime support for parallelism must add additional synchronization to potentially racy program executions. One kind of synchronization, biased reader–writer locks, provides performance advantages, but its lock ownership transfers incur expensive coordination among threads.

2.1 Capturing Cross-Thread Dependences

This paper focuses on *runtime support* for parallelism, which we define generally as any analysis or system that tracks or enforces an execution's multithreaded behavior. Runtime support needs to *capture* cross-thread dependences (i.e., write–read, write–write, and read–write data dependences involving two threads), which means doing one of the following:

Tracking (detecting) cross-thread dependences: Example kinds of runtime support include data race detectors (e.g., [22, 23]), atomicity checkers (e.g., [5, 24]), and dependence recorders for deterministic multithreaded record & replay (e.g., [10, 35]).

Controlling (enforcing) cross-thread dependences Example runtime support includes transactional memory (e.g., [27, 58]), enforcement of memory models (e.g., [41, 47]), and deterministic execution (e.g., [4, 40]).

For data-race-free (DRF) executions, capturing cross-thread dependences requires instrumenting only *synchronization* operations, since DRF0-based memory models guarantee serializability of synchronization-free regions for DRF executions [1, 2, 8, 36].

However, programs routinely have intentional and unintentional data races (e.g., [35]). Thus, runtime support must instrument each access that might be involved in a data race.[1] Furthermore, to ensure that dependences are captured soundly, runtime support must 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. Straightforward lock implementations use atomic operations and memory fences, which incur remote cache misses and serialize in-flight instructions (e.g., [23, 24, 34, 35, 48]).

We note that some approaches have sidestepped these challenges but have other limitations. For example, record & replay can avoid tracking dependences by relying on replication and speculation, but its performance relies on extra available cores [52]. Some STMs avoid acquiring a lock for every accessed object in a transaction, but sacrifice scalability as a result (e.g., [19]).

2.2 Biased Reader–Writer Locks

Prior work introduces so-called *biased* locks, in which each object's lock is "biased" toward one thread, or multiple threads in the case of reader locks [10, 13, 30, 31, 43, 45, 46, 53]. These thread(s) can reacquire the lock *without* using an atomic operation. However, in order for a thread T to acquire a lock owned by other thread(s), T must *coordinate* with the other thread(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.

[1] Even after applying sound static data race detection as a filter, many accesses cannot be proven to be DRF [16, 22, 35, 54].

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

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.

Without loss of generality, this paper builds on one design of biased reader–writer locks (which is most closely based on prior work called *Octet* [10]), which we now describe. Each object's lock can have any of the following states: $\mathsf{WrEx}_{\mathsf{T}}$ (write-exclusive for thread T), $\mathsf{RdEx}_{\mathsf{T}}$ (read-exclusive for T), or $\mathsf{RdSh}_{\mathsf{c}}$ (read-shared for all threads, subject to a counter c that helps detect dependences from the last write soundly [10]). Instrumentation at each program store and load "acquires" a write or read lock on the accessed object's lock, as the following pseudocode shows:

```
if (obj.lockState != WrEx_T) {
  /* slow path: acquire obj.lockState for write */
}
obj.f = ...; // program store

...

if (obj.lockState != WrEx_T &&
    obj.lockState != RdEx_T &&
    (obj.lockState != RdSh_c || T.rdShCount < c)) {
  /* slow path: acquire obj.lockState for read */
}
... = obj.f; // program load
```

When a thread's instrumentation tries to acquire an object's lock in a state that conflicts with the lock's current state (e.g., at a read by T2 to an object whose lock is in $\mathsf{WrEx}_{\mathsf{T1}}$ state), the thread initiates a *coordination protocol* to ensure that the lock's current owner thread(s) relinquish the lock safely, as described next.

2.3 Synchronous Coordination

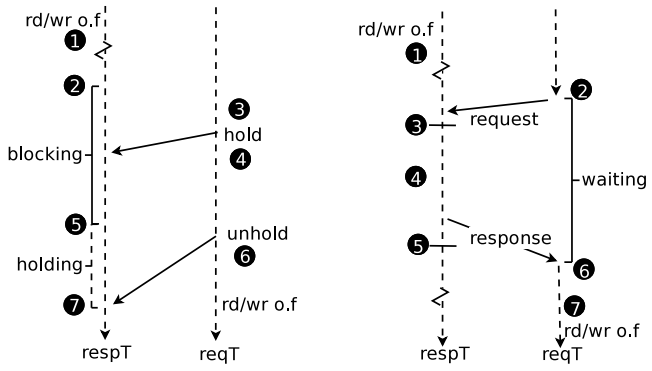
When a thread wants to acquire a biased lock that it does not already own, it must *coordinate* with the owner thread(s) to ensure they acknowledge the ownership transfer. We refer to prior work's coordination [10, 13, 30, 31, 43, 45, 46, 53] as *synchronous coordination*, to distinguish it from the asynchronous coordination that this paper introduces.

Next we briefly overview how synchronous coordination works. Suppose a thread, called the *requesting thread*, reqT, wants to acquire an object'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 $\mathsf{WrEx}_{\mathsf{respT}}$ or $\mathsf{RdEx}_{\mathsf{respT}}$ 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.

First, reqT atomically changes the state of the object's lock to an *intermediate* state, $\mathsf{RdEx}^{\mathsf{Int}}_{\mathsf{reqT}}$ or $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{reqT}}$, depending on whether a read or write lock is needed. The intermediate state simplifies the protocol by allowing only one thread at a time to perform a conflicting transition on an object's lock.

In order to preserve instrumentation–access atomicity, respT participates in the protocol only when it is at a *safe point*: a program point that is definitely *not* in the middle of instrumentation or its corresponding access. If respT is performing a *blocking* operation—such as waiting to acquire a program lock or waiting for I/O—then respT is at a *blocking safe point*, and reqT makes an *implicit* request to respT. Figure 1(a) shows coordination using an implicit request.

If respT is instead actively executing program code, then reqT must wait for respT to reach a safe point and respond. reqT performs an *explicit* request by adding a request to respT's *request queue*. Figure 1(b) shows the details of coordination using an explicit request. respT responds once it reaches a safe point. Safe



(a) **Implicit request**: (1) respT accessed o previously. (2) respT enters a blocked state before performing some blocking operation. (3) reqT wants to access o. It changes o's lock's state to $\mathsf{RdEx}^{\mathsf{Int}}_{\mathsf{reqT}}$ or $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{reqT}}$. (4) reqT performs runtime-support-specific actions while keeping respT in a "blocked and held" state. (5) respT finishes blocking but waits until hold(s) have been removed. (6) reqT removes the hold on respT and changes o's lock's state to $\mathsf{WrEx}_{\mathsf{reqT}}$ or $\mathsf{RdEx}_{\mathsf{reqT}}$ and proceeds to access o. (7) respT leaves the "blocked and held" state.

(b) **Explicit request**: (1) respT accessed an object o previously. (2) reqT wants to access o. It changes o's lock to $\mathsf{RdEx}^{\mathsf{Int}}_{\mathsf{reqT}}$ or $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{reqT}}$, 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 $\mathsf{WrEx}_{\mathsf{reqT}}$ or $\mathsf{RdEx}_{\mathsf{reqT}}$ and proceeds to access o.

Figure 1. Details of the *synchronous* coordination protocol.

points include not only blocking safe points but also *yield points*, which are regularly executed program points that various existing language implementations already provide (e.g., for timely joining of parallel garbage collection). While reqT waits for a response, it is considered to be at a blocking safe point, so other threads can perform implicit requests with reqT acting as a *responding* thread, avoiding deadlock. Finally, respT responds to reqT's request.

As our results show (Section 7), 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 7.1 describes our overall experimental methodology):

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

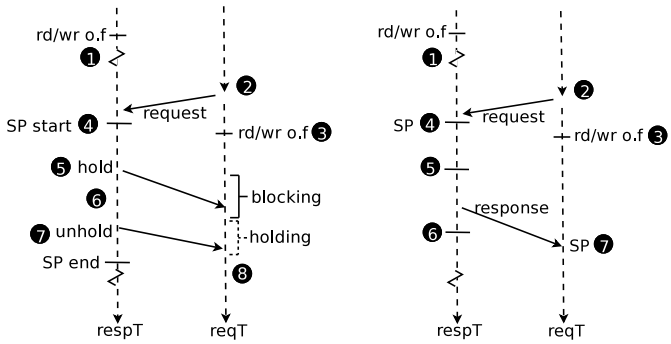
Coordination is substantially more expensive than a same-state check because coordination requires several memory accesses, including atomic operations and memory fences. On average, coordination is more than an order of magnitude more costly if it uses an explicit than an implicit request, since an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

3. Asynchronous Coordination

This section describes our novel approach for *asynchronous coordination* (AC) that allows a thread to continue executing while it waits for coordination to complete. We first describe how the *AC protocol* works, followed by how threads perform *asynchronous accesses* that overlap with coordination (the key technical challenge).

3.1 The Asynchronous Coordination Protocol

In AC, a requesting thread does *not* wait for responses after sending requests. Thus, a requesting thread receives responses at some later



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

Figure 2. Details of the *asynchronous* coordination protocol.

point in its execution, and, a requesting thread can have outstanding requests for multiple objects simultaneously. To support this functionality, the AC protocol differs from the synchronous 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, the AC protocol extends synchronous 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 2 shows how the AC protocol works. respT 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 2(a), by first putting reqT into a "blocked and held" state 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 2(b) shows, by adding adding a response to reqT's queue. Once reqT reaches a safe point, it changes the object's lock's state.

We note that the AC protocol differs from the synchronous coordination protocol for *explicit* requests only. When coordination uses an *implicit* request, it follows the same steps as in Figure 1(a).

Figure 2 shows a single requesting thread sending requests to respT. In general, multiple requesting threads might 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. But for a transition from RdSh to $\mathsf{WrEx}_{\mathsf{reqT}}$, reqT may need to wait for multiple responses. The protocol maintains a count 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.

Thus, the AC 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 is processed at the receiving thread's next safe point.

3.2 Handling Asynchronous Accesses

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

3.2.1 Asynchronous Stores

A thread T performs an asynchronous store by *deferring* the store, buffering the location and new value in T's *store buffer*. The intuition behind deferring stores is that another thread may be simultaneously (racily) accessing the same location, so performing the store directly could cause a cross-thread dependence to be missed. Once T gets exclusive ownership of 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 asynchronous stores by T to objects locked in $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T}}$ state. (We found that supporting asynchronous stores to other lock states provided little benefit.)

Preserving program semantics. Deferring program stores changes program behavior since other threads can read out-of-date values from the affected memory locations. However, DRF0-based memory models (including the Java and C++ memory models) allow substantial reordering of operations, except across synchronization operations [1, 2, 8, 36], 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).

On the other hand, why would it be *incorrect* to allow stores to be deferred *past* synchronization release operations? Figure 3 shows an example. While object o's lock is intially in $\mathsf{WrEx}_{\mathsf{T1}}$, T1 sets o.f's value to v1. Next, T2 attempts to store v2 to o.f. T2 changes o's lock to $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T2}}$, sends an asynchronous request to T1, and defers the store in its store buffer, avoiding conflicting with any accesses by T1, which still "owns" o. T2 must commit its buffered store before executing unlock(l). Otherwise, T1's load after its lock(l) operation might read the old value v1, violating semantics, since communicating synchronization guarantees visibility [8, 36].

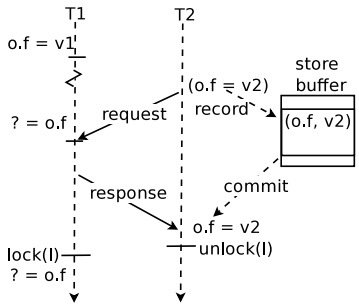


Figure 3. Asynchronous stores cannot be deferred past release operations.

3.2.2 Asynchronous Loads

At an asynchronous load by T to an object o, T first checks whether the same location has already been buffered in T's store buffer. (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 instead of loading from memory.

Otherwise, thread T performs the load from memory normally. An asynchronous load thus does *not* affect program semantics: the execution still conforms to the memory model (e.g., performing the load would be permitted in the original program). However, an asynchronous load could certainly affect the ability 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.

AC thus handles each asynchronous load by *logging the loaded value* in a *runtime-support-specific way*. The intuition is that logging the value enables runtime support to handle all values resulting from potentially uncaptured cross-thread dependences. For example, our asynchronous dependence recorder logs the value in order to assist replay (Section 4), and our asynchronous 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 asynchronous store (i.e., every entry in its store buffer). This restriction limits AC'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 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*. While it may be possible to defer various kinds of release operations (e.g., thread fork), we focus on deferring only 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 asynchronous stores that executed before the lock release (based on order-tracking bookkeeping).

This behavior naturally preserves program semantics because a thread continues to hold a lock while it waits for responses for asynchronous stores, effectively expanding the lock's critical section—making other threads wait and thus increasing lock contention.

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 AC 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 asynchronous coordination (AC) to optimize a dependence recorder. Recording dependences efficiently is the key challenge of multithreaded record & replay, which enables both reproducing rare bugs and replicating multithreaded processes (e.g., [11, 35, 52]). However, recording dependences is expensive due to the high cost of tracking dependences between all potentially shared memory accesses [34, 35].

4.1 Synchronous Dependence Recorder

Prior work builds a dependence recorder, which we call the *synchronous recorder*, on top of biased reader–writer locks that use synchronous coordination [10]. It records all happens-before

edges [33] established by the biased reader–writer locks. Another execution can replay these happens-before edges deterministically.

The synchronous recorder records happens-before edges for every kind of lock state transition. Here we describe only conflicting state transitions that initiate coordination with an explicit request, since AC modifies only that case.

Preliminaries. Recording a happens-before edge involves recording both its *source* and its *sink*. Each of these 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 and exit and loop back edge.

Recording a happens-before edge for coordination that uses an explicit request. When a requesting thread triggers synchronous coordination with an explicit request, it waits for the responding thread to reach a safe point and respond. Referring back to Figure 1(b) on page 3, the recorder records the happens-before edge from point #4 on respT to point #6 on reqT. At point #4, respT records the source of the edge by writing into its log (1) the DPL of the current safe point into its record log and (2) the number of responses made so far, incremented at every response. At point #6, reqT records the edge's sink by writing into the log (1) the DPL of the pending program access and (2) the number of responses sent so far by respT (this value is observed racily, but it is guaranteed to be large enough to capture the edge soundly, and small enough to avoid deadlock during replay). During replay, reqT waits at the recorded DPL for respT to send the same number of responses.

4.2 Asynchronous Dependence Recorder

Our *asynchronous recorder* extends the synchronous recorder by using AC instead of synchronous coordination. The asynchronous recorder allows asynchronous 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 2 on page 3 for examples of happens-before edges recorded by the asynchronous 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. If the replayed execution replays these same edges, it will not necessarily get the same behavior.

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 asynchronous 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 2(a) and Figure 2(b)). A replayed execution can reuse this value to ensure determinism.

5. Software Transactional Memory

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

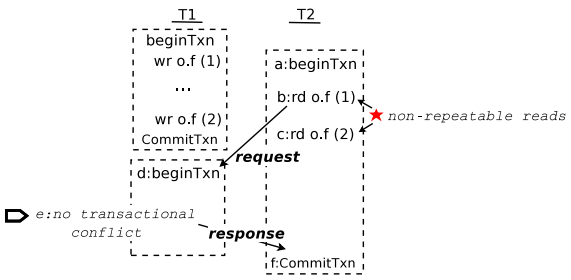


Figure 4. Allowing asynchronous accesses in transactions would lead to serializability violations. The values in parentheses after each executed store and load are the values written and read, respectively.

5.1 Synchronous STM

Prior work introduces an STM that uses biased reader–writer locks that employ synchronous coordination [58]. We call this STM the *synchronous STM*. The synchronous 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 using an *explicit* response. 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 Asynchronous STM

Unless handled properly, asynchronous loads and stores could be unable to detect and resolve transactional conflicts. Figure 4 shows an example problematic execution. Thread T2 performs two asynchronous loads (at b and c) 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 (at d) that has *not* accessed o, so T1 accurately reports no transactional conflict (at e). However, the result is unserializable because T2's loads see different values. Another problematic issue (not shown) is that performing asynchronous transactional stores directly could lead to unserializable results due to another thread loading the value simultaneously.

Our *asynchronous STM* addresses these issues as follows. In the asynchronous STM, each asynchronous, transactional load logs its loaded value, and *validates* the value later. Each asynchronous, 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 asynchronous loads and perform all deferred stores.

Asynchronous loads. A requesting thread reqT performs an asynchronous 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) (e.g., at f in Figure 4). 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.)

Asynchronous stores. A requesting thread reqT defers an asynchronous store by buffering its location and value in the store buffer, which is analogous to the *redo log* used by STMs that use lazy versioning [28]. 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 AC design, our asynchronous STM does *not* support loads by T to objects locked in intermediate states *other than* $\mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Int}}$ and $\mathsf{RdEx}_{\mathsf{T}}^{\mathsf{Int}}$; 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 before validating reads.

Guaranteeing progress. The *synchronous* STM guarantees progress by detecting all conflicts eagerly and then aborting the younger transaction [51, 58]. However, the *asynchronous* STM cannot guarantee progress, since any transaction that fails read validation must abort. Other mixed-mode STMs have similarly lacked progress guarantees [29, 44]. Standard techniques such as exponential backoff can help to alleviate livelock. However, for the asynchronous STM, a simple (but as-yet-unimplemented) solution exists: if a transaction repeatedly fails read validation, it reexecutes *synchronously*, guaranteeing it will commit (once it becomes oldest, at least).

Semantics. Lazy read validation can lead to so-called *zombie* transactions whose behavior is impossible in any serializable execution [28]. In managed languages such as Java, zombies are not a serious problem because memory and type safety are preserved [18, 38]. Targeting a native language such as C++ would require additional support to provide *sandboxing* of zombie transactions [18]. On other STMs, zombie transactions can get stuck in infinite loops that are impossible in any serializable execution. The asynchronous STM cannot experience this issue because asynchronous reads get validated within a bounded amount of time.

Comparison with prior work. Most STMs employ either entirely lazy or entirely eager concurrency control (e.g., [19, 21, 51, 58]). Some STMs combine lazy and eager mechanisms, by using eager concurrency control for *writes* and lazy validation for *reads* [29, 44]. The asynchronous 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. Implementation

We have implemented AC and the asynchronous recorder and STM in Jikes RVM, a Java virtual machine (JVM) [3] that performs competitively with commercial JVMs [6]. Our AC implementation builds on the publicly available *Octet* implementation of biased reader–writer locks [10]. Our asynchronous recorder builds on the synchronous recorder that is part of the Octet implementation [10]. Our asynchronous STM builds on the publicly available synchronous STM called *LarkTM* [58]. Our implementations reuse features of the synchronous implementations as much as possible.

7. Evaluation

This section evaluates the performance and run-time characteristics of runtime support using asynchronous coordination (AC), compared with synchronous approaches.

7.1 Methodology

Benchmarks. The experiments execute the following benchmarks:

- The DaCapo benchmarks, versions 2006-10-MR2 and 9.12-bach (2009) [7], excluding single-threaded programs and programs that Jikes RVM cannot execute.
- Fixed-workload versions of SPECjbb2000 and SPECjbb2005.[3]
- The Java Grande benchmarks (excluding microbenchmarks) [50].

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.

7.2 Evaluating Asynchronous Coordination

This section evaluates AC and compares it with synchronous coordination, by executing instrumentation that captures dependences but performs no runtime support on top of it.

Measuring the problem. We first measure the cost of synchronous coordination, as well as the maximum benefit that can be obtained from optimizations. Figure 5 shows runtime overhead of three configurations over an unmodified JVM. These configurations all instrument accesses to capture dependences using biased reader–writer locks. The first configuration, *SC*, uses *synchronous coordination* (as described in Section 2.3) and adds 140% 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 capturing dependences using biased reader–writer locks.

Ideal is an *unsound* configuration that eliminates most of synchronous coordination's cost. In this configuration, after a thread sends an explicit request, it simply continues execution without waiting for any response. Responding threads in turn simply ignore requests. This configuration attempts to estimate an upper bound on the performance that AC might be able to provide. On average, it adds 76% overhead—a little more than half of the overhead added by the sound configuration. The remaining costs are due to instrumentation at every access that perform same-state transitions and other state transitions, including conflicting transitions that trigger coordination using implicit requests. In addition, even though requesting threads do not wait for responses and responding threads ignore requests, *Ideal* incurs remote cache misses by sending explicit requests. *Ideal* slightly outperforms baseline execution in a few cases, for reasons we are still investigating.

AC's effectiveness at hiding coordination costs. Figure 5's last configuration, *AC (no stall at unlock)*, uses asynchronous coordination with the second optimization in Section 3.3. Its overhead over baseline execution is 91%, a significant reduction (49% relative to baseline execution time) from synchronous coordination's 140% overhead. Furthermore, AC achieves much of the maximum possible benefit, approaching *Ideal*'s 76% overhead.

We note that for sparse, AC significantly outperforms the *Ideal* configuration. This unintuitive 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 AC might provide. As we show later in this section, AC can actually reduce the number of conflicting transitions (compared with synchronous coordination); AC reduces sparse's conflicting transitions substantially, leading to significantly lower overhead than for *Ideal*.

Figure 6 shows the same configurations as Figure 5 plus two other AC configurations. This graph is a *speedup* graph (higher is better), normalized to *SC*.

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

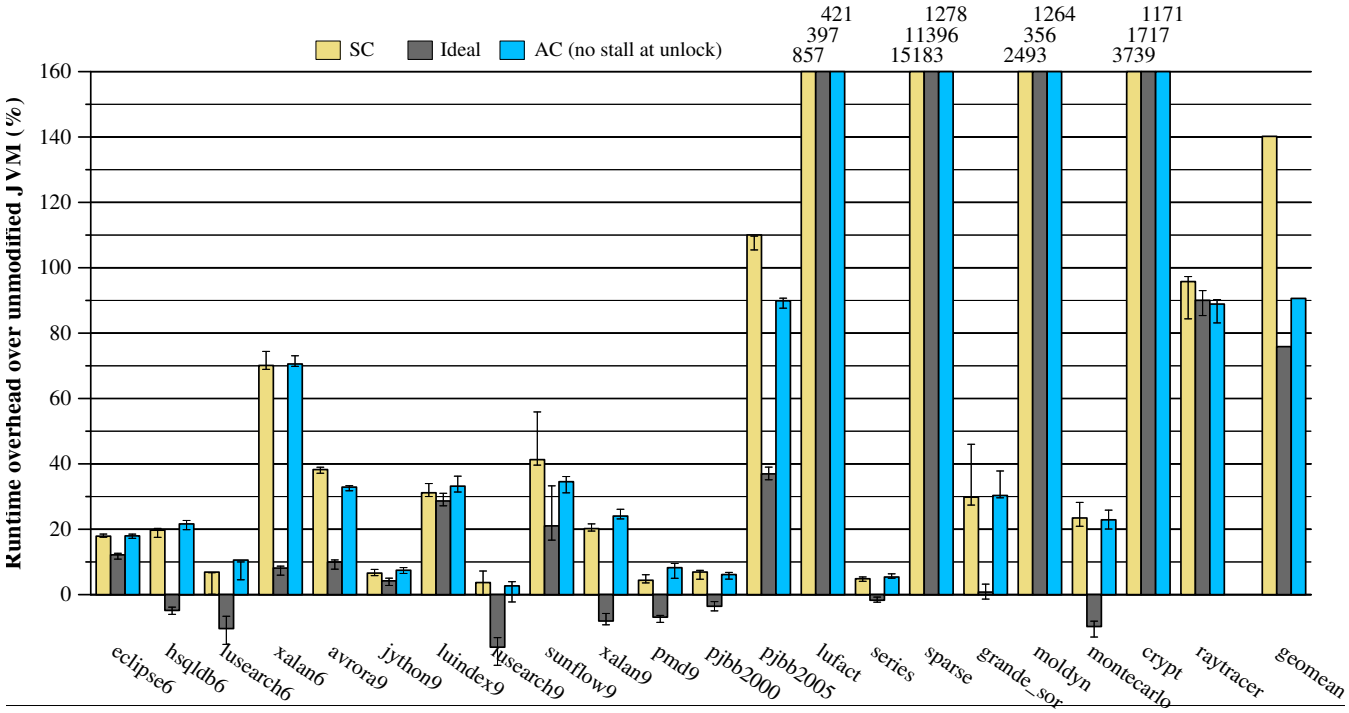


Figure 5. Run-time overhead added by capturing dependences using (1) synchronous coordination, compared with (2) an ideal, unsound configuration that eliminates coordination latency and (3) AC.

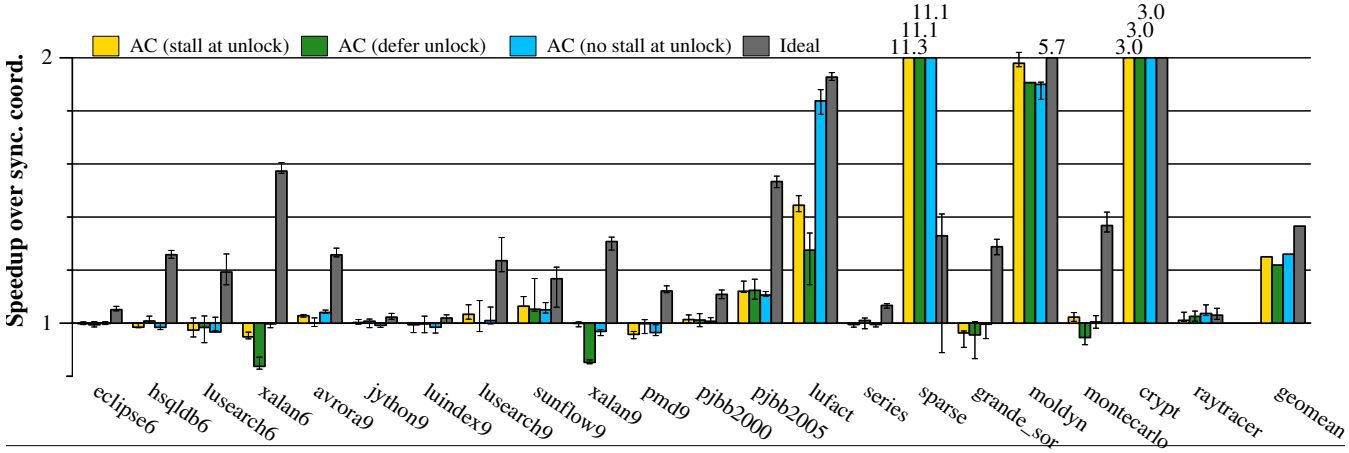


Figure 6. Speedup of AC related to synchronous coordination. *Ideal* is an unsound configuration that provides an upper bound on AC's performance.

The AC configurations, which are all sound, are the following:

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

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

AC (no stall at unlock): The second optimization described in Section 3.3 (and also shown in Figure 5). At a lock release, a thread continues execution even if there are outstanding asynchronous stores. However, no thread except T can read from an object locked in $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T}}$ state.

The three AC configurations each provide an average speedup of 1.22–1.26X over synchronous coordination. These speedups are close to the average speedups achieved by *Ideal* (1.36X), suggesting that AC is getting most of the maximum possible benefit from hiding the cost of coordination due to explicit requests.

While the AC configurations outperform *SC* and get close to *Ideal*'s performance on average, AC does not help much with the gap between *SC* and *Ideal* in several cases (hsqldb6, xalan6, avrora9, lusearch9, xalan9, grand_sor, and montecarlo). As we show later, AC changes the balance of explict versus implicit requests triggered by coordination (relative to synchronous coordination), since AC causes threads to spend more time executing code instead of blocking at safe points. This change cancels out AC's potential performance benefits in several cases, and it represents a challenge for future work. Another significant source of AC overhead is bookkeeping costs: its queue reprentation leads to more costs than synchronous coordination's, and keeping track of asynchronous events and accesses requires performing additional work.

The *AC (defer unlock)* configuration does not improve performance on average (nor significantly for any indvidual program) compared with the *AC (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 *AC (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 *AC (no stall at unlock)* provides almost no average benefit over *AC (stall at unlock)*.

Run-time characteristics. Next we focus on understanding factors contributing to the performance difference between AC and synchronous coordination. Table 1 reports runtime statistics of capturing dependences with AC and synchronous coordination. The table uses the AC configuration *AC (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 AC sometimes reduces *how many* conflicting transitions occur, relative to synchronous coordination. 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 synchronous coordination, 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 AC—particularly when executing past release operations as permitted by the *AC no stall at unlock* configuration—a thread is more likely to perform consecutive *asynchronous* accesses to a contended object, leading to fewer conflicting transitions, compared with synchronous coordnation. This effect is directly responsible for AC 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 synchronous coordination overhead and can benefit more from AC.

The *Coord. responses* columns tally AC 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 asynchronous accesses. While some of these accesses immediately follow coordination requests, others are repeat accesses to the same memory location. Yet others are loads from objects for which some *other* thread has initiated coordination. Due to these cases, asynchronous accesses can outnumber conflicting transitions. On the other hand, conflicting transitions can outnumber asynchronous accesses, since an implicit request does not lead to an asynchronous access.

7.3 Performance of Runtime Support

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

Dependence recorder. Figure 7 shows the performance of the synchronous and asynchronous recorders. (We exclude crypt and raytracer, since the recorders fail to execute them successfully.) Not surprisingly, the performance story for the recorders is similar to the story for tracking dependences alone. On average, the asynchronous recorder is 1.16X faster than the synchronous recorder.

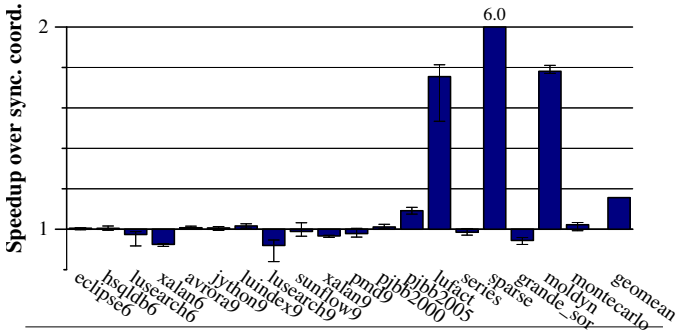


Figure 7. Run-time speedup of the asynchronous dependence recorder over the synchronous dependence recorder.

We note that although AC can reduce the cost of coordination, the asynchronous recorder still needs to record each happens-before edge. Some asynchronous loads can avoid conflicting transitions and coordination entirely, but the recorder must log each value of such loads. The asynchronous recorder thus often logs *more* than the synchronous recorder by every asynchronous load's value and every asynchronous store's commit point. Notably, the asynchronous recorder's log size is about 2X the synchronous recorder's for xalan6, xalan9, and sparse; and about 7X for grande_sor. For other programs, the asynchronous recorder logs less than 50% more than the synchronous recorder.

Software transactional memory. This section compares the synchronous and asynchronous STMs using the *STAMP* benchmark suite [15], ported to Java by other researchers [20, 32]. The Java port includes six working benchmarks.

Figure 8 shows the execution time of the asynchronous and synchronous STMs. We first note that both STMs typically scale poorly after 8 threads; prior work has also found that STAMP has limited scalability [59]. Furthermore, our platform has 8 threads per core, leading to greater inter-thread communication for >8 threads.

For genome and vacation, AC 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, leading to fewer opportunities for AC to improve performance. For vacation, the ratio of implicit to explicit requests stays fairly constant across thread counts, but AC's benefit diminishes because accesses per thread decrease as threads increase, leading to less latency per thread for AC to reduce.

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

For kmeans, intruder, and ssca2, AC provides sustained or increasing benefit over synchronous coordination for 8 to 32 threads. For these programs, the asynchronous STM achieves a significantly lower rate of aborting transactions than the synchronous STM. The asynchronous STM validates asynchronous loads at object field and array element granularity, as opposed to the synchronous 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 asynchronous loads. However, direct benefits from AC are limited because many transactions are short, and the asynchronous STM must wait at transaction end for all outstanding asynchronous accesses (Section 5.2).

In summary, AC reduces the cost of capturing dependences significantly, particularly for programs with high coordination costs, recovering much of the maximum possible performance from eliminating cooridnation latency entirely. AC's benefit is limited by correctness constraints (e.g., limitations of deferring past synchronization) and indirect effects (e.g., increases to the explicit-to-implicit request ratio when using AC compared with synchronous coordination). Although the asynchronous recorder and STM suffer draw-

	Synchronous coordination				Asynchronous coordination							
	Lock state transitions		Coord. requests		Lock state transitions		Coord. requests		Coord. responses		Async. 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 synchronous and asynchronous coordination.

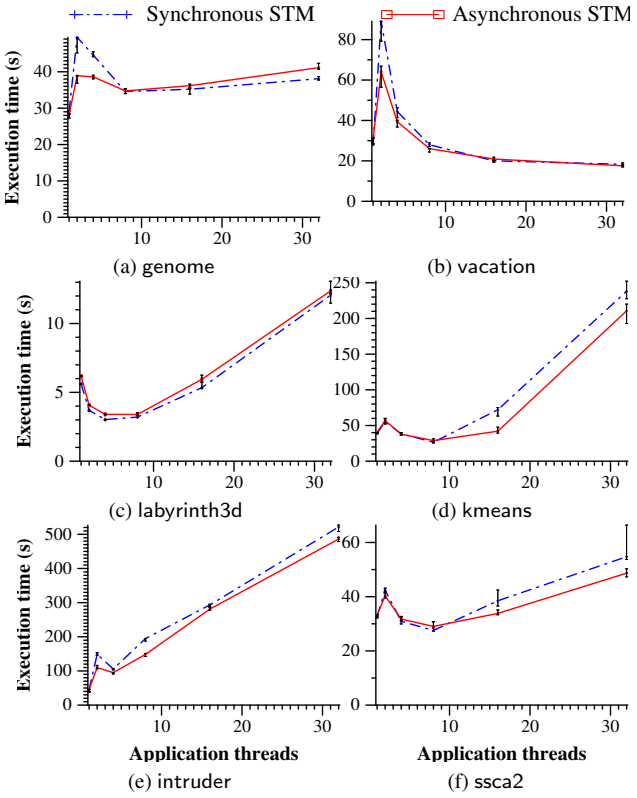


Figure 8. Performance of the synchronous and asynchronous STMs.

backs (increased log size and transactional correctness constraints, respectively) that limit the improvement somewhat, our experiment shows that AC helps optimize runtime support's performance.

8. Related Work

Section 2 covered the closest related work. This section compares asynchronous coordination to other approaches.

Handling coordination costs. Some prior work targets the high cost of coordination in the context of providing instrumentation–access atomicity using biased locking. Cao et al. hybridize biased and unbiased locking [14]. Von Praun and Gross use a model that switches to an unbiased state for shared objects [53], but atomic operations required for unbiased states lead to high overhead [10].

Biased program locks. This paper's work uses biased locking to provide instrumentation–access atomicity for capturing cross-thread dependences. Other work has used biased locking for program locks [31, 43]. 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 best-effort, limited hardware support [57]. TSX could potentially solve the same problem as software-based reader–writer locks, although recent work suggests that TSX struggles to outperform even atomic-operation-based synchronization [37, 42, 57]. In contrast, biased reader–writer locks avoid atomic operations, although an empirical comparison is beyond this paper's scope.

9. Conclusions

Runtime support based on biased reader–writer locks can achieve significantly lower overhead than traditional locks, but conflicting accesses require threads to coordinate with each other. Asynchronous coordination (AC) seeks to overlap coordination latency with useful program work, using a novel approach that preserves correctness. AC provides modest benefit overall since (1) many programs incur low coordination costs to begin with; (2) AC cannot hide all costs, particularly the cost of remote cache misses; and (3) correctness requirements limit asynchronony at program synchronization release operations and transaction commits. This work demonstrates that opportunities exist to exploit the flexibility of program semantics and the mechanics of sound runtime support, leading to more practical runtime support for parallel programs.

References

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

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

[3] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research

Virtual Machine Project: Building an Open-Source Research Community. *IBM Systems Journal*, 44:399–417, 2005.

[4] 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.

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

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

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

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

[9] 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.

[10] 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.

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

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

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

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

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

[16] 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.

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

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

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

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

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

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

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

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

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

[26] 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.

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

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

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

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

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

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

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

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

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

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

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

[38] 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.

[39] 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.

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

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

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

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

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

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

[51] 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.

[52] 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.

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

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

[55] 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.

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

[57] 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.

[58] 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.

[59] 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.