Design, Verification and Applications of a New Read-Write Lock Algorithm

Jun Shirako
Rice University
6100 Main Street
Houston, Texas 77005
shirako@rice.edu

Nick Vrvilo
Rice University
6100 Main Street
Houston, Texas 77005
nick.vrvilo@rice.edu

Eric G. Mercer
Brigham Young University
3334 TMCB
Provo, Utah 84602
eric.mercer@byu.edu

Vivek Sarkar
Rice University
6100 Main Street
Houston, Texas 77005
vsarkar@rice.edu

ABSTRACT

Coordination and synchronization of parallel tasks is a major source of complexity in parallel programming. These constructs take many forms in practice including directed barrier and point-to-point synchronizations, termination detection of child tasks, and mutual exclusion in accesses to shared resources. A read-write lock is a synchronization primitive that supports mutual exclusion in cases when multiple reader threads are permitted to enter a critical section concurrently (read-lock), but only a single writer thread is permitted in the critical section (write-lock). Although support for reader threads increases ideal parallelism, the read-lock functionality typically requires additional mechanisms, including expensive atomic operations, to handle multiple readers. It is not uncommon to encounter cases in practice where the overhead to support read-lock operations overshadows the benefits of concurrent read accesses, especially for small critical sections.

In this paper, we introduce a new read-write lock algorithm that reduces this overhead compared to past work. The correctness of the algorithm, including deadlock freedom, is established by using the Java Pathfinder model checker. We also show how the read-write lock primitive can be used to support high-level language constructs such as object-level isolation in Habanero-Java (HJ) [6]. Experimental results for a read-write microbenchmark and a concurrent SortedLinkedList benchmark demonstrate that a Java-based implementation of the proposed read-write lock algorithm delivers higher scalability on multiple platforms than existing read-write lock implementations, including ReentrantReadWriteLock from the java.util.concurrent library.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

SPAA'12, June 25–27, 2012, Pittsburgh, Pennsylvania, USA.

Keywords

Read-write locks, mutual exclusion, model checking.

1. INTRODUCTION

It is widely recognized that computer systems anticipated in the 2020 timeframe will be qualitatively different from current and past computer systems. Specifically, they will be built using homogeneous and heterogeneous many-core processors with 100's of cores per chip, their performance will be driven by parallelism, and constrained by energy and data movement [15]. This trend towards ubiquitous parallelism has forced the need for improved productivity and scalability in parallel programming models. One of the major obstacles to improved productivity in parallel programming is the complexity of coordination and synchronization of parallel tasks. Coordination and synchronization constructs take many forms in practice and can be classified two types, *directed* and *undirected*. Directed synchronization, such as termination detection of child threads and tasks using join, sync [4], and finish [5] operations, collective synchronization using barriers and phasers [16], point-to-point synchronization using semaphores, and data-driven tasks [17] semantically define a "happens-before" execution order among portions of the parallel programs. Directed synchronization is most often used to enable deterministic parallelism. Undirected synchronization, such as operations on locks, actors, and transactional memory systems, is used to establish mutual exclusion in accesses to shared resources with strong or weak *atomicity* guarantees [9, 10]. Undirected synchronization is most often used to enable data-race-free nondeterministic parallelism.

Recent efforts to improve software productivity for atomicity are focused on declarative approaches that let the programmer demarcate blocks of codes to be atomically executed and defer the complexity of maintaining atomicity to the compiler and the runtime system. This approach aims to support higher programmability while delivering performance comparable to well-tuned implementations based on fine-grained locks. On the other hand, primitive fine-grained locks are still in demand for performance-critical software such as commonly used runtime libraries and system software rather than user programs. Further, fine-grained locks are easy to support across a wide range of languages and platforms, while the high-level declarative approaches are limited to specific languages and/or require special hardware support.

A *read-write lock* enforces a special kind of mutual exclusion, in

which multiple reader threads are allowed to enter a critical section concurrently (read-lock), but only a single writer thread is permitted in the critical section (write-lock). Although support for reader threads increases ideal parallelism, the read-lock functionality usually requires additional mechanisms, including expensive atomic operations to handle multiple readers. It is not uncommon to find cases where the overhead of supporting read-lock operations overshadows the benefit of concurrent read accesses. As a result, the use of read-write locks can often degrade overall performance relative to standard fine-grained locks, due to the overheads involved.

In this paper, we introduce a new read-write lock algorithm that reduces the overhead of read-write locks compared to existing implementations. In practice, the additional overhead of our read-lock mechanism is equivalent to a pair of atomic increment and decrement operations invoked when a reader thread enters and leaves the critical section. As shown in our experimental results on a 64-thread Sun UltraSPARC T2 system and a 32-core IBM Power7 system, using efficient implementation of atomic increment operations, such as java.util.concurrent.AtomicInteger [13], achieves significant performance improvement over existing read-write lock approaches, including Java's ReentrantReadWriteLock. Although the implementation in this paper is based on atomic integers in Java, the underlying read-write lock algorithm is easily implemented in any language on any platform that includes an atomic compare-and-swap operation.

The correctness of the algorithm, including deadlock freedom, is established by using the Java Pathfinder model checker. We also show how the read-write lock primitive can be used to support high-level language constructs such as object-level isolation in Habanero-Java (HJ) [6]. Experimental results for a read-write micro-benchmark and a concurrent SortedLinkedList benchmark demonstrate that a Java-based implementation of the proposed read-write lock delivers higher scalability on multiple platforms than existing read-write lock implementations, including Java's ReentrantReadWriteLock.

The rest of the paper is organized as follows. Section 2 uses a sorted-list example as motivation for read-write locks by showing how they can be used directly in the example, or indirectly to support HJ's object-level isolation construct. Section 3 describes the details of our new read-write lock algorithm, and Section 4 proves the correctness of the proposed algorithm. Section 5 presents our experimental results, and Section 6 and Section 7 summarize related work and our conclusions.

2. USE OF READ-WRITE LOCKS IN A SORTED LIST ALGORITHM

2.1 Sorted Linked List Example with Explicit Locks

In this section, we use the Sorted Linked List example from [7] as a motivation for read-write locks, since it is representative of linked lists used in many applications. The nodes in the list are sorted in ascending order based on their integer key values (parameter `v` in Listing 1). There are four list operations: `insert`, `remove`, `lookup`, and `sum`. It is assumed that no lock is needed for `lookup`, fine-grained locking suffices for `insert` and `remove`, and a coarse-grained lock is needed for `sum`.

We implement this example by using a single read-write lock (`globalRWLock`), and multiple standard fine-grained locks (one per node). The assumption is that multiple calls to `insert` and `remove` can execute in parallel if they operate on disjoint nodes, but none of those calls can execute in parallel with a call to `sum`; also, it is always safe for `lookup` to execute in parallel with any other operation. The inherent ordering in a linked list structure can be used to avoid deadlock when acquiring fine-grained locks on the nodes.

As shown in Listing 1, `insert(v)` atomically inserts a new node with value `v` into the list if no node in the list has value `v`. The role of `globalRWLock` is to manage the mutual exclusion rules for `insert`, `remove`, and `sum` operations. Specifically, `insert` and `remove` obtain read-locks on `globalRWLock`, thereby ensuring that they can all execute in parallel with each other. However, `sum` acquires a write-lock on `globalRWLock` to ensure that no instance of `insert` and `remove` can execute in parallel with it.

Since the new node corresponding to `v` must be atomically inserted between nodes `prev` and `curr`, `insert` operation obtains the locks corresponding to these nodes (lines 9–10 and 14–15) after obtaining a read-lock on `globalRWLock` . `remove(v)` (not shown in Listing 1) has a similar structure to `insert`, and atomically removes the node with value `v` if such a node exists in the list. Therefore, `remove` also obtains locks for the node with value `v` and its previous node. `sum()` computes the sum of all the values in the list, after obtaining the write-lock for `globalRWLock`.

2.2 Sorted Linked List Example with Object-Based Isolation in Habanero-Java

Section 2.1 showed how to implement the functionality required for `insert`/`remove` and `sum`, by using a global read-write lock and local locks for nodes. This two-level locking approach can also be used to enable higher levels of abstraction and safety, as described in this section. We briefly introduce Habanero-Java's *isolated* construct [6, 10] to support mutual exclusion with global and object-based (local) isolation, and show how a read-write lock can be employed to implement this extension.

- **Object-based isolation:** The `isolated(`⟨*obj-set*⟩`)` ⟨*stmt*⟩ construct supports mutual exclusion on ⟨*stmt*⟩ with respect to the objects specified in ⟨*obj-set*⟩. Mutual exclusion between two statements ⟨*stmt1*⟩ and ⟨*stmt2*⟩ is guaranteed if and only if ⟨*obj-set1*⟩ and ⟨*obj-set2*⟩ have a non-empty intersection. Further, while *isolated* statements may be nested, an inner *isolated* statement is not permitted to acquire an object that wasn't already acquired by an outer *isolated* statement.

- **Global isolation:** The `isolated(*)` ⟨*stmt*⟩ construct expands the scope to all objects and support global mutual exclusion on ⟨*stmt*⟩. This is the default semantics for HJ's *isolated* construct if no object set is provided

Unlike Java's *synchronized* construct, this definition of object-based isolation is guaranteed to be implemented with deadlock freedom. Further, no reference in the object set can trigger a NullPointerException as in Java's *synchronized* construct. Finally, Java does not have a *synchronized(*)* statement analogous to `isolated(*)`.

To rewrite the `insert` method in Listing 1 with object-based isolation, lines 13–15 can be replaced by "`isolated(lk1,lk2) {`", and lines 22–24 can be replaced by "`}`". Likewise, the `sum` method can be rewritten using an `isolated(*)` construct. These rewrites result in much simpler code since the programmer does not have to worry about deadlock avoidance or null checks on the objects (a null entry is simply an empty contribution to the object set).

The actual implementation of object-based isolation relies on the use of a global read-write lock, as illustrated in Section 2.1. Deadlock avoidance is obtained by using some `Comparable` field so as to order the objects. Effectively, a program written using object-based isolation will be translated to code that is quite similar to the explicit lock version in Listing 1 .

```
1  public boolean insert(int v) {
2    while (true) {
3      INode curr, prev = null;
4      for (curr = first; curr != null; curr = curr.getNext()) {
5        if (curr.getValue() == v) return false;  // v already exists
6        else if (curr.getValue() > v) break;
7        prev = curr;
8      }
9      OrderedLock lk1 = getLocalLock(prev);      // Get local locks corresponding to nodes
10     OrderedLock lk2 = getLocalLock(curr);
11
12     boolean set = false;
13     globalRWLock.read_lock();                  // Obtain global read lock
14       if (lk1 != null) lk1.lock();             // Obtain local locks for nodes
15       if (lk2 != null) lk2.lock();
16       if (validate(prev, curr)) {
17         INode neo = new INode(v);
18         link(prev, neo, curr);
19         assignLocalLock(neo);                  // Assign a local lock to the new node
20         set = true;
21       }
22       if (lk2 != null) lk2.unlock();           // Release local locks for nodes
23       if (lk1 != null) lk1.unlock();
24     globalRWLock.read_unlock();                // Release global read lock
25     if (set) return true;
26 } }
27
28 public int sum() {
29   int s = 0;
30   globalRWLock.write_lock();                   // Obtain global write lock
31     for (INode curr = first; curr != null; curr = curr.getNext())
32       s += curr.getValue();
33   globalRWLock.write_unlock();                 // Release global write lock
34   return s;
35 }
```

Listing 1: SortedLinkedList insert and sum methods

3. PROPOSED LOCK APPROACH AND ALGORITHM

This section first introduces an OrderedLock algorithm (Section 3.1) that is based on queue lock approaches such as Ticket Lock [11], Anderson's array-based queue lock [1], and Partitioned Ticket Lock [3]. We then introduce the OrderedReadWriteLock algorithm (Section 3.2), the primary focus of this paper. Although support for reader threads in read-write lock increases ideal parallelism, the read-lock functionality in current implementations typically requires additional mechanisms for managing multiple readers, which often overshadow the benefit of concurrent read accesses. In our approach, we focus on reducing the overhead of read-lock operations in OrderedReadWriteLock, while the overhead of write-lock operation is comparable to that of general queue lock operations such as OrderedLock. As described below, OrderedLock and OrderedReadWriteLock do not use atomic operations other than the atomic increment/decrement operation, which can be implemented with a compare-and-swap primitive.

3.1 OrderedLock

Our OrderedLock algorithm consists of two steps: 1) determine the synchronization order among critical sections on a first-come-first-served basis (as in Ticket Locks [11]) and 2) perform a point-to-point synchronization between two sections with continuous orders using arrays as in Anderson's array-based queue lock [1]. Figure 1 shows a sample lock-unlock sequence for four threads operating on a single OrderedLock. The first step needs an atomic increment operation to determine the synchronization order of each section (1st to 5th in Figure 1), and the second step preserves the order by array-based point-to-point synchronization. This description of the OrderedLock is roughly equivalent to the Partitioned Ticket Lock [3], although we discuss some further performance tuning in Section 5.1.1. Listing 2 summarizes the OrderedLock interface. Note that the `lock` operation returns the `order` of the requester in the queue, and the `unlock` operation uses `order`. Our expectation is that the interface in Listing 2 will be used by library and language implementers, rather than application programmers who are not expected to see the internal details of the `order` values.

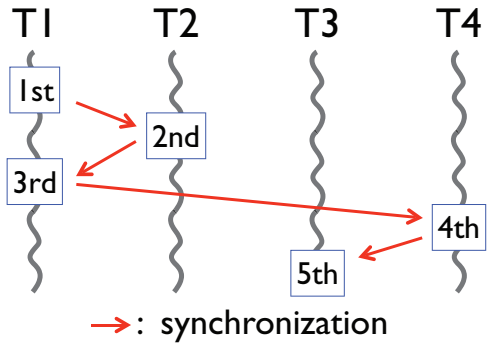

Figure 1: OrderedLock by four threads

```
1 class OrderedLock {
2   int lock();                // Acquire the lock and return order in queue
3   void unlock(int order);    // Use order when performing an unlock operation
4 }
```

Listing 2: Interface of queue lock

3.2 OrderedReadWriteLock

The write lock/unlock operation in OrderedReadWriteLock is equivalent to general lock/unlock, and the read lock/unlock operation consists of the following steps.

- **Read-lock**
  - First reader: Invoke general lock `OrderedLock.lock` to obtain lock
  - Other readers: Wait for the first reader to obtain lock
- **Read-unlock**
  - Last reader: Invoke general unlock `OrderedLock.unlock` to release lock
  - Other readers: No-op

There are several policy decisions on the priority for obtaining lock. In our approach, both a read-lock operation (reader) and a write-lock operation (writer) have the same priority when the write-lock is released (fair reader-writer policy), and once a read-lock is obtained, the subsequent readers can share the read lock regardless of the presence of waiting writers. Figure 2 shows an example for OrderedReadWriteLock by four threads T1, T2, T3, and T4. The writers, for example, W:1st and W:4th by T1, require general mutual exclusion and their own unique synchronization orders, while a synchronization order is assigned to several different readers, such as R:2nd (by T1 and T2), and R:3rd (by T2, T3, and T4). Here R:2nd by T1 is the last reader to releases the lock, and then R:3rd by T4 becomes the next first reader. As shown below, the key function of OrderedReadWriteLock is the management of the unlocking process among concurrent readers, whose efficiency directly affects the overall synchronization performance.

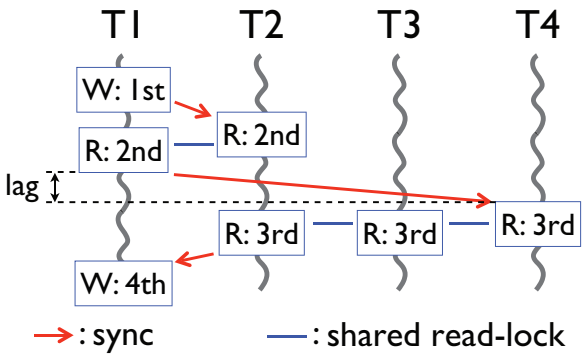

Figure 2: OrderedLock and OrderedReadWriteLock by four threads

Listing 3 describes OrderedReadWriteLock in written Java. Methods `write_lock` and `write_unlock` are equivalent to general `lock` and `unlock` of OrderedLock (lines 9 and 10). At the beginning of methods `read_lock` and `read_unlock`, atomic increment and decrement operations are respectively performed so as to detect the first reader and last reader (lines 13 and 28). For method `read_lock`, the first reader invokes `OrderedLock.lock` to obtain a synchronization order `readOrder`, which is shared by all readers (line 14) and set `readLocked = true` to notify that the order has been obtained (line 15), while non-first readers wait for the first reader to obtain the order (line 22). For method `read_unlock`, the last reader sets `readLocked = false` to notify that the current lock is released (line 32) and invokes `OrderedLock.unlock` to release the lock.

It is possible that `read_lock` is invoked by one thread while another is in the midst of the unlocking process in `read_unlock`. We refer to readers that invoke `read_lock` during an unlocking transition as *rapid readers*. In that case a first rapid reader will be blocked by `OrderedLock.lock` at line 14 as normal. In contrast, non-first rapid readers will reach line 17 and have two scenarios: 1) if flag `transit` is `true` (set at line 29), the rapid reader is blocked at line 19 until the last reader completes the unlocking process, or 2) if flag `transit` is `false`, the rapid reader passes the blocking operations (lines 19 and 22) and enters the critical section. The code at line 31 is to manage rapid readers encountering the second scenario so that the last reader waits for the rapid readers to leave the critical section. Section 4 discusses the correctness of OrderedReadWriteLock. Note that this transition process is a rare occurrence, at least in our benchmarks, and has almost no affect on performance as shown in Section 5.

The OrderedReadWriteLock provides no fairness or progress guarantees. Any number of readers can enter, exit, and re-enter their critical sections under the same lock as long as `numReaders` remains positive, thus enabling an infinite stream of repeated readers to starve a writer.

4. VERIFICATION OF LOCK ALGORITHM WITH MODEL CHECKING

We prove the correctness of our OrderedReadWriteLock implementation in three stages. We start by proving correctness for a four-thread model where each thread performs a single locking operation, assuming sequential consistency. We then prove a lack of data races for the same model, which guarantees sequential consistency. Finally, we show the results obtained from our four-thread model with single operations are sufficient to generalize to any number of threads and locking operations. The underlying OrderedLock is assumed correct throughout the proof due to its simplicity and basis in previous work (see Section 3.1).

For the first stage of our proof we use the Java Pathfinder (JPF) model checker, an automatic verification tool for concurrent Java programs [18]. JPF takes as input the Java program to verify, an environment to provide program input, and correctness conditions in the form of program assertions in the environment or model. JPF then uses state space exploration to enumerate all possible sequentially consistent executions allowed by the program and environment. The result is an exhaustive proof showing the absence of any execution that violates an assertion. JPF employs a partial order reduction to reduce the number of executions it must consider in the

```
1  class OrderedReadWriteLock {
2    OrderedLock olock = new OrderedLock();    // General lock
3    int readOrder = 0;                        // Synchronization order shared by readers
4    volatile boolean readLocked = false;      // True if first reader obtained lock
5    AtomicInteger numReaders;                 // # readers at this moment (init by 0)
6    volatile boolean transit = false;         // True if last reader is releasing lock
7    AtomicInteger numBlockedInTransit;        // # blocked readers in transition (init by 0)
8
9    int write_lock() {  return olock.lock();  }
10   void write_unlock(int myOrder) {  olock.unlock(myOrder);  }
11
12   int read_lock() {
13     if (numReaders.addAndGet(1) == 1) {     // First reader
14       readOrder = olock.lock();             // Get sync order to be shared by readers
15       readLocked = true;                    // Notify that lock has been obtained
16     } else {
17       if (transit) {                        // Transition of unlocking process (mostly false)
18         numBlockedInTransit.addAndGet(1);
19         while (transit);                    // Wait for last reader to release lock
20         numBlockedInTransit.addAndGet(-1);
21       }
22       while (!readLocked);                  // Wait for first reader to obtain lock
23     }
24     return readOrder;
25   }
26
27   void read_unlock(int myOrder) {
28     if (numReaders.addAndGet(-1) == 0) {    // Last reader
29       transit = true;                       // Start transition of unlocking process
30       if (numReaders.get() > 1)             // Manage rapid readers (mostly false)
31         while (numBlockedInTransit.get()+1 < numReaders.get());
32       readLocked = false;                   // Notify that lock is released
33       transit = false;                      // End transition of unlocking process
34       olock.unlock(myOrder);                // Actually release lock
35 } } }
```

Listing 3: OrderedReadWriteLock

proof construction and mitigate state space explosion in the verification [12]. The key aspects of the environment for checking our OrderedReadWriteLock are shown in Listing 4.

Each of the four threads in our model is an instance of EnvThread shown in Listing 4. The library call in the `switch` on line 9 tells JPF to explore the execution resulting for each value in the inclusive range 0 to 1, allowing our model to cover all possible combinations of readers and writers. We check for correct mutual exclusion using two assertions. Line 13 ensures that the shared-counter value does not change while under a read lock and that the `readLocked` is always set while a reader is in the critical section. Since `readLocked` is set immediately after a reader obtains the lock and unset immediately before the last reader releases the lock, this assertion guarantees that the lock is held by a reader, not a writer. Line 22 ensures that the final count reflects the actual number of writes to the shared counter. JPF checks these assertions for all possible thread orderings. Additionally, termination on all execution paths proves freedom from deadlock.

To prove data-race freedom we use Java Racefinder (JRF), a JPF module for detecting data races [8]. It is important to distinguish between a race condition and a data race. A race condition is anytime there are two concurrent accesses to a shared variable. A data race, however, is when those accesses conflict, such as a read and a write, and are not *happens-before ordered* [14]. Happens-before orderings are induced by synchronization primitives, such as atomic read-modify-write operations, access to variables declared volatile, explicit locks, etc. JRF tracks these synchronization primitives to construct the happens-before relation on-the-fly during state space exploration to prove a program is data-race free. For the data-race verification, we use the same model as with JPF, thus proving that there are no data races in the scenario or in the underlying lock. The Java memory model guarantees sequential consistency in the absence of data races [14]; therefore, the assumption in our JPF model of sequentially consistent executions is correct.

LEMMA 4.1. *The OrderedReadWriteLock is free of deadlock, implements mutual exclusion, and is free of data-race for up to four threads with each thread obtaining a lock.*

PROOF. Exhaustive proof via model checking with JPF and JRF using the environment described in Listing 4. The running time for each verification run is under 20 minutes on a standard desktop machine. Full details with all source and test harness files to recreate the proof are available online.[1] For completeness, we also mutated the lock and verified via JPF that the lock fails as expected. □

Lemma 4.1 proves correctness for our four-thread environment. We now prove that the model described in Listing 4 using four threads is sufficient to conclude correct behavior of the lock for any number of threads. We prove this by showing that any thread added in addition to the four used in our model will not result in exploring any interesting new states.

THEOREM 4.2. *The OrderedReadWriteLock is free of deadlock, implements mutual exclusion, and is free of data-race for any number of threads each obtaining any sequence of locks.*

[1] `http://www.cs.rice.edu/~nv4/papers/spaa2012/ORWLockTest.tgz`

```
1  volatile int sharedCounter = 0;
2  AtomicInteger envCounter = new AtomicInteger();
3  OrderedReadWriteLock instance = new OrderedReadWriteLock();
4  private java.util.Random generator = new java.util.Random();
5
6  class EnvThread extends Thread {
7     public void run() {
8        int next, mine;
9        switch(generator.nextInt(2)) {
10          case 0: // Reader thread
11             next = instance.read_lock();
12             mine = sharedCounter;
13             assert(mine == sharedCounter && instance.readLocked);
14             instance.read_unlock(next); break;
15          case 1: // Writer thread
16             envCounter.getAndAdd(1);
17             next = instance.write_lock();
18             sharedCounter += 1;
19             instance.write_unlock(next); break;
20 }  }  }
21
22 assert(sharedCounter == envCounter.get());
```

Listing 4: Generic environment for depth-bounded model check of the OrderedReadWriteLock using JPF.

PROOF. The underlying lock, `olock` in Listing 3, is assumed correct and orders requests to its lock interface accordingly. As such, it is sufficient to prove the case of a single writer with more than three readers because multiple writers are arbitrated by the `olock`, and anything less than four threads is covered by Lemma 4.1. All line numbers refer to Listing 3 in the proof.

Consider the case where the readers do not transfer ownership of `olock`. In such a scenario, either the writer or a single reader holds `olock` (line 9 and line 14). By Lemma 4.1, a single writer with three readers is correct as either the writer or readers will block until the other finishes. If the writer finishes first, then only the last reader enters line 28, does not enter the if-statement on line 30 as it is the last reader, and eventually releases `olock` on line 34. This case is no different than the all-readers case in Lemma 4.1.

Consider now the case where the readers transfer ownership of `olock`. As before, if the writer holds `olock`, then the readers block and the problem reduces to Lemma 4.1 with multiple readers. Let us then assume that the writer is queued up to obtain `olock`, which is currently held by a reader. Let us further assume that one reader takes the true branch on line 28 of Listing 3. We will refer to this thread as $reader_A$. Next, another reader takes the true branch on line 13 of `read_lock`, attempting to reacquire the lock. We will refer to this thread as $reader_B$. A final thread, which we will refer to as $reader_C$, is then forced to take the false branch on line 28 since `numReaders` is greater than 1. In this scenario, additional writer or reader threads do not affect the lock behavior, and the problem reduces to that of Lemma 4.1 with one writer and three readers.

To be specific, having `numReaders` > 1 means that $reader_C$ is now free to take any path through the remainder of the `read_lock` method. Since all remaining conditions in the method depend entirely on the current state of $reader_A$ in `read_unlock` and the state of `olock` (i.e., whether a writer or reader obtains `olock` next), we can conclude by Lemma 4.1 that all remaining control paths through `read_lock` are covered by $reader_C$, and no additional readers or writers are required to elicit new behavior. All additional threads added to the environment will continue to access the lock, read or increment the counters, and pass the assertions in our environment as in the four-thread scenario.

We have shown that we can obtain full coverage of the lock's behavior with the four threads in our counter scenario. Any threads interacting with the lock in excess of four will only duplicate behavior already observed in the four-thread model. Therefore, Theorem 4.2 is true via Lemma 4.1. These properties hold barring integer overflow in the internal lock state and improper client use of the lock. □

5. EXPERIMENTAL RESULTS

In this section, we present experimental results for the proposed OrderedReadWriteLock. All results in this paper were obtained on two platforms. The first platform is a 64-thread (8 cores $\times$ 8 threads/core) 1.2 GHz Sun UltraSPARC *T2* system with 32 GB main memory running Solaris 10. We conducted all experiments on this system by using the Java 2 Runtime Environment (build 1.5.0_12-b04) with Java HotSpot Server VM (build 1.5.0_12-b04, mixed mode). The second platform is a 32-core 3.55 GHz IBM *Power7* system with 256 GB main memory running Red Hat Enterprise Linux release 6.1. We used the IBM J9 VM (build 2.4, JRE 1.6.0) for all experiments on this platform. On both platforms, the main Java program was extended with a 10-iteration loop within the same process, and the best result was reported so as to reduce the impact of JIT compilation time and other JVM services in the performance comparisons.

5.1 Summary of implementation

In this section, we briefly summarize our preliminary Java-based implementations of OrderedLock and OrderedReadWriteLock.

5.1.1 OrderedLock

As described in Section 3.1, our general lock implementation is based on Ticket Lock [11] and has the same array-based extension as Partitioned Ticket Lock [3] so as to reduce memory and network contention. Ticket Lock consists of two counters, *request counter* to contain the number of requests to acquire the lock and *release counter* to contain the number of times the lock has been released. A thread requesting a lock is assigned a ticket (the value of the request counter) and waits until the release counter equals its ticket. In the implementation, the request counter must be an atomic vari-

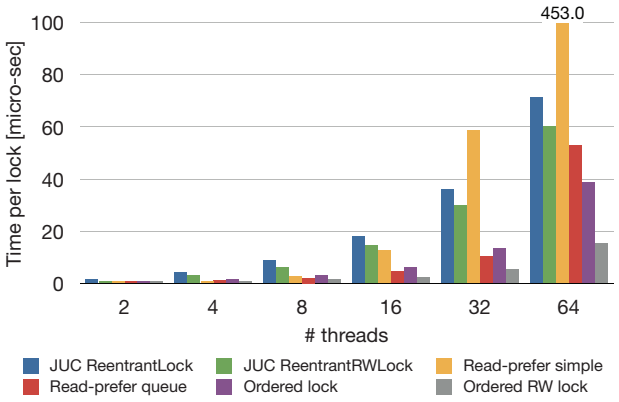

Figure 3: ReadWrite SyncBench (read rate = 90% on T2)

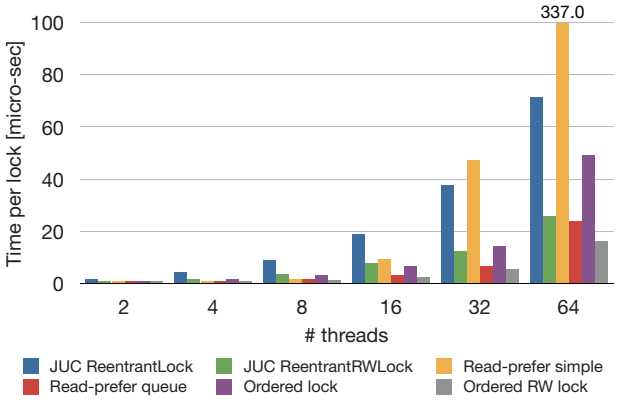

Figure 4: ReadWrite SyncBench (read rate = 99% on T2)

able so that concurrent threads can get unique tickets, while the release counter can be a non-atomic variable since it is always updated by a thread that is releasing the lock. To avoid the contention on the single global release counter, Partitioned Ticket Lock and OrderedLock employ similar extensions to Anderson's array-based queue lock [1], which replaces the release counter variable by an array with cache line padding so that different threads can access different elements of the array. Appendix A includes a pseudo-code summary of our OrderedLock implementation in Java.

In an attempt to improve the scalability of atomic increments for the request counter, OrderedLock employs the idea of adding a delay to the atomic updating loop if the update fails [1]. By adding a delay, we reduce the contention and bus traffic for the update on the atomic variable. The delay function has various choices in the implementation, such as random, proportional, exponential, and constant. In this paper, we used a random function of the form, `delay * (1.0 + rand.nextDouble())`, where `delay` is a tunable parameter for each platform and `rand` is an instance of java.util.Random that returns a double value between 0 and 1.

5.1.2 OrderedReadWriteLock

The Java implementation for OrderedReadWriteLock was based on the design discussed earlier in Listing 3. We also employ the delay optimization for atomic increment operations discussed in Section 5.1.1.

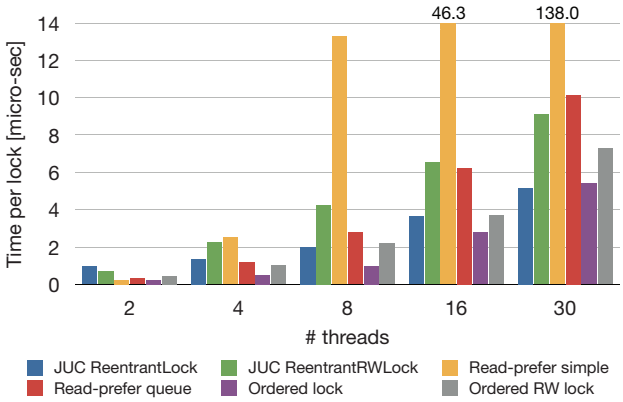

Figure 5: ReadWrite SyncBench (read rate = 90% on Power7)

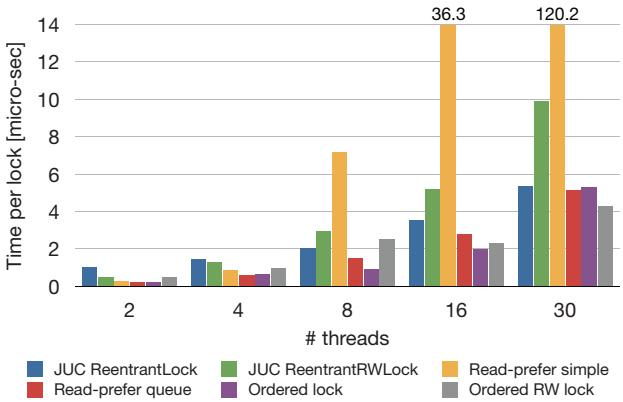

Figure 6: ReadWrite SyncBench (read rate = 99% on Power7)

5.2 Microbenchmark Performance

This section presents synchronization performance using a microbenchmark. We use the JGFSyncBench microbenchmark with the following extension to evaluate read-write lock functionality. Given an arbitrary number of parallel threads, each thread randomly invokes `readWork` with the probability of *reading_rate* or `writeWork`, whose probability is (1 - *reading_rate*). `readWork` and `writeWork` are guarded by read-lock and write-lock, respectively. Figures 3–6 show the synchronization performance (time per operation) on *T2* and *Power7* when *reading_rate* is 90% and 99%. The number of parallel threads ranges from 2 to 64 on *T2* and 2 to 30 on *Power7*. [2] There are six experimental variants:

- **JUC ReentrantLock** is the general lock implementation of java.util.concurrent (JUC).
- **JUC ReentrantRWLock** stands for ReentrantReadWriteLock, which is the read-write lock of JUC.
- **Read-prefer simple** is an implementation of the simple reader-preference lock approach [11]. [3]

[2] On *Power7*, two threads are reserved due to the possible system workload.

[3] We selected the read-preference policy because of the benchmarks that contain many read-locks and few write-locks.

Table 1: Rate of transitions over total critical sections executed

	read rate = 90%	read rate = 99%
T2 with 8 threads	7.5×10^{-5}	7.1×10^{-5}
T2 with 64 threads	9.7×10^{-5}	2.3×10^{-5}
Power7 with 8 threads	70.4×10^{-5}	22.6×10^{-5}
Power7 with 32 threads	4.4×10^{-5}	5.9×10^{-5}

- **Read-prefer queue** is an implementation of the scalable reader-preference queue-based lock [11].
- **OrderedLock / OrderedRWLock** is the proposed general / read-write lock approach.

Figure 3 shows the synchronization performance when the *reading_rate* = 90% on *T2*, which demonstrates that OrderedReadWriteLock gives much better efficiency than other lock approaches, by the factor of 4.67× for JUC ReentrantLock, 3.94× for JUC ReentrantReadWriteLock, 29.61× for simple reader-preference lock, 3.46× for scalable reader-preference queue-based lock, and 2.54× for OrderedLock. Figure 4 shows the case where the *reading_rate* is increased to 99% on *T2*. More concurrent reader threads improve the performance of other read-write lock approaches, although OrderedReadWriteLock wins in all experimental variants.

Figures 5 and 6 show the synchronization performance on *Power7* when the *reading_rate* = 90% and 99%, respectively. Due to faster clock frequency on *Power7*, the overlapping work in the readers' critical sections is relatively small compared to *T2*. Therefore, general lock implementations of ReentrantLock and OrderedLock attain better performance than read-write locks when the *reading_rate* = 90%. In the case where the *reading_rate* = 99%, however, OrderedReadWriteLock performs better by a factor of 1.25× than JUC ReentrantLock, 2.29× than JUC ReentrantReadWriteLock, 1.24× than OrderedLock, and 1.19× than scalable reader-preference queue-based lock.

Regarding the overhead due to the unlocking transition process discussed in Section 3.2, we measured the frequency of this process in *T2* and *Power7*. Specifically, we measured the ratio of the total number of last readers delayed by rapid readers (the condition at line 30 in Listing 3 becomes `true`) to the total number of critical sections executed under the lock. As shown in Table 1, the extremely low transition frequency indicates a negligible overhead.

5.3 Application Performance with Read-write Lock

We used SortedLinkedList to demonstrate the two-level lock approach shown in Section 2. We supported the `insert`, `remove` and `sum` operations using the following implementation variants.

- **Reentrant single** uses a JUC ReentrantLock as the single global lock to guarantee mutual exclusion of `insert`, `remove`, and `sum`.
- **Reentrant 2-lv** employs the two-level lock approach using JUC ReentrantReadWriteLock and JUC ReentrantLock.
- **Ordered single** uses a OrderedLock as the single global lock.
- **Ordered 2-lv** employs the two-level lock approach, using OrderedReadWriteLock and JUC ReentrantLock.
- **All ordered 2-lv** employs the two-level lock approach, using OrderedReadWriteLock and OrderedLock.

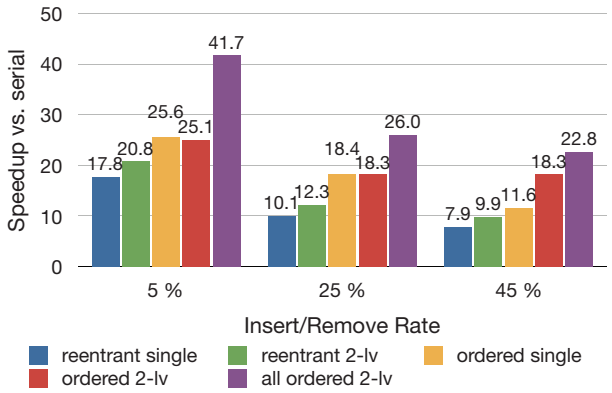

Figure 7: Speedup for SortedLinkedList 64-thread T2

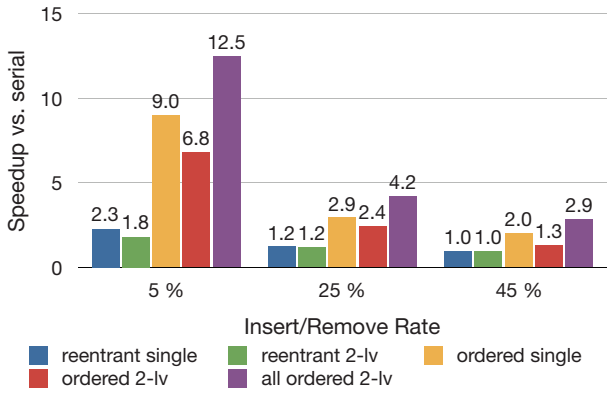

Figure 8: Speedup for SortedLinkedList 32-core Power7

Given an arbitrary number of parallel threads, each thread randomly invokes one of four list operations, `insert(v)`, `remove(v)`, `lookup(v)`, and `sum()`, where the value of `v` is a random number. The range of `v`, which determines the maximum list length, is 0 to 2048. There are 256 local locks to handle all nodes, and therefore up to 8 nodes are mapped into a local lock. We used a simple lock-assignment scheme for linked-list nodes based on a uniform sub-range partitioning of the node values. The probability of `insert` and `remove` is given by *insert_remove_rate*, the probability of `sum` is fixed as 1%, and `lookup` has the remaining possibility of (1 - *insert_remove_rate* × 2 - 0.01).

Figures 7 and 8 show the speedup ratio relative to the sequential execution on *T2* and *Power7*, respectively. The number of parallel threads is 64 on *T2* and 30 on *Power7*. Comparing **reentrant 2-lv** and **ordered 2-lv**, OrderedReadWriteLock performs better than ReentrantReadWriteLock for all cases by a factor of up to 3.78×, even though both implementations employ ReentrantLock as the local lock. Moreover, **all ordered 2-lv** shows that the combination of OrderedReadWriteLock and OrderedLock always performs the best, up to 41.7× speedup on *T2* and 12.5× speedup on *Power7*.

6. RELATED WORK

There is an extensive literature on general and read-write locks approaches. In this section, we focus on a few past contributions that are most closely related to this paper.

The FIFO queue-based lock is a simple and efficient approach to support mutual exclusion. Ticket Lock [11] is a queue-based lock that consists of two counters, one containing the number of requests to acquire the lock and the other the number of times the lock has been released. A thread requesting lock is assigned a ticket (the value of the request counter) and waits until the release counter equals its ticket.

Memory and network contentions can occur when all threads continuously check the same release counter. To reduce this contention and improve scalability, several array-based queue-lock approaches that allow threads to check different locations (different cache lines) have been proposed [1, 3]. List-based queue locks, such as MCS [11] and CLH [2] locks, also support scalable synchronizations in a similar manner and require a smaller space size.

Extending queue-based locks, several read-write lock algorithms with reader-preference, writer-preference, and fair reader-writer policies have been proposed [11]. Although these read-write locks employ efficient local-only spinning implementations, the processes to support read-write lock functionality require lots of atomic operations such as compare_and_store, fetch_and_add, fetch_and_and, fetch_and_or, and fetch_and_store because their algorithms strictly preserve the orders in their FIFO queues.

7. CONCLUSION

In this paper, we introduced a new read-write lock algorithm that supports concurrent reader threads and has lower overhead than existing implementations. The algorithm lowers overhead by tracking reader counts and only allowing the first and last reader threads to interact with the underlying lock that implements mutual exclusion between the readers and writers. The lower overhead is not free, however, as the new algorithm is considerably more complex than other existing algorithms. We demonstrated the correctness of this new algorithm, including deadlock freedom, via the Java Pathfinder model checker. To demonstrate the utility of this new lock, we described how the read-write lock primitive can support high-level language constructs, such as object-level isolation in Habanero-Java (HJ) [6]. We further implemented the proposed read-write lock algorithm as a Java library and demonstrated the efficiency of the approach on two platforms. The experimental results for a read-write microbenchmark show that our algorithm performs 3.94× and 2.29× better than java.util.concurrent.ReentrantReadWriteLock on a 64-thread Sun UltraSPARC T2 system and 32-core IBM POWER7 system, respectively. Performance measurements for a concurrent SortedLinkedList benchmark also demonstrate higher scalability for our algorithm on multiple platforms over the benchmark set. Opportunities for future research include scalability evaluations on a wider range of benchmark programs, support for additional high-level language constructs using the proposed read-write lock (*e.g.*, read/write permission regions), and experimenting with its implementation in non-Java language environments.

8. ACKNOWLEDGMENTS

We are grateful to John Mellor-Crummey and William Scherer at Rice University, and Doug Lea at SUNY Oswego, for their feedback on this work and its relationship to past work on read-write locks. The SortedList example used to obtain the results reported in this paper was derived from an earlier HJ version implemented by Rui Zhang and Jisheng Zhao. We would like to thank Jeff Bascom at Brigham Young University for developing the initial JPF model and generating early verification results, and Jill Delsigne at Rice University for her assistance with proof-reading the final version of this paper.

This work was supported in part by the U.S. National Science Foundation through awards 0926127 and 0964520. We would like to thank Doug Lea for providing access to the UltraSPARC T2 system used to obtain experimental results for this paper. The POWER7 system used to obtain experimental results for this paper was supported in part by NIH award NCRR S10RR02950 and an IBM Shared University Research (SUR) award in partnership with CISCO, Qlogic, and Adaptive Computing.

9. REFERENCES

[1] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. In *Proc. IEEE Int'l. Parallel and Distributed Processing Symp. (IPDPS)*, January 1990.

[2] T. Craig. Building FIFO and priority-queueing spin locks from atomic swap. In *Technical Report TR 93-02-02*. University of Washington, Dept. of Computer Science, 1993.

[3] D. Dice. Brief announcement: A partitioned ticket lock. In *SPAA '11: Proceedings of the 23rd annual ACM symposium on parallelism in algorithms and architectures*, New York, NY, USA, 2011. ACM.

[4] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM.

[5] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, May 2009. IEEE Computer Society.

[6] Habanero Java (HJ) Project. http://habanero.rice.edu/hj, 2009.

[7] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM Press.

[8] K. Kim, T. Yavuz-Kahveci, and B. A. Sanders. Precise data race detection in a relaxed memory model using heuristic-based model checking. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 495–499, Washington, DC, USA, 2009. IEEE Computer Society.

[9] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[10] R. Lublinerman, J. Zhao, Z. Budimlić, S. Chaudhuri, and V. Sarkar. Delegated Isolation. In *OOPSLA '11: Proceedings of the 26th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, 2011.

[11] J. Mellor-Crummey and M. Scott. Algorithms for Scalable Synchronization on Shared Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[12] NASA Ames Research Center. JPF developer guide: On-the-fly partial order reduction. http://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/partial_order_reduction, 2009.

[13] T. Peierls, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java*

Concurrency in Practice. Addison-Wesley Professional, 2006.

[14] W. Pugh. JSR-133: Java memory model and thread specification. http://www.jcp.org/en/jsr/detail?id=133, August 2004.

[15] V. Sarkar, W. Harrod, and A. E. Snavely. Software Challenges in Extreme Scale Systems. January 2010. Special Issue on Advanced Computing: The Roadmap to Exascale.

[16] J. Shirako et al. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 277–288, New York, NY, USA, 2008. ACM.

[17] S. Taşırlar and V. Sarkar. Data-Driven Tasks and their Implementation. In *ICPP'11: Proceedings of the International Conference on Parallel Processing*, Sep 2011.

[18] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engg.*, 10:203–232, April 2003.

APPENDIX

A. ORDEREDLOCK IMPLEMENTATION

```
1  class OrderedLock {
2    AtomicInteger order =new AtomicInteger(0);
3    // Array size (equal to HW threads)
4    int arraySize = getNumHardwareThreads();
5    VolatileInt[] syncVars = new VolatileInt[
         arraySize];
6
7    int lock() {
8      // Atomic increment
9      int myOrder = order.getAndAdd(1);
10     // Compute corresponding index
11     int idx = Math.abs(myOrder % arraySize);
12     VolatileInt sv = syncVars[idx];
13     // Spin-lock on myOrder
14     while (sv.val != myOrder);
15
16     return myOrder;
17   }
18
19   void unlock(int myOrder) {
20     int next = myOrder + 1;
21     // Compute corresponding index
22     int idx = Math.abs(next % arraySize);
23     VolatileInt sv = syncVars[idx];
24     // Release spin-lock on (myOrder+1)
25     sv.val = next;
26   }
27
28   class VolatileInt {
29     volatile int val;
30     // Avoid false sharing
31     int pad1, pad2, ..., padN;
32   }
33 }
```

Listing 5: OrderedLock

Listing 5 provides pseudo code for our Java-based implementation of OrderedLock. For method `lock`, an atomic increment operation determines the synchronization order `myOrder` (line 9), followed by a point-to-point waiting process on `myOrder` (lines 11–14). The waiting process first computes the index value `idx` to access array `syncVars` based on `myOrder` (line 11) and waits until the `val` field equals `myOrder` (line 14). The value of `myOrder` is returned since it is used for the unlocking process (line 16). Method `unlock` works as a point-to-point signal operation on `myOrder+1` so as to release the spin-lock of the following lock operation (lines 20–25). The signaling process also computes `idx` for `next = myOrder + 1` in the same manner (lines 20–22), and sets `next` to its `val` field (line 25). As shown at line 5, `syncVars` is an array of `VolatileInt` class that contains padding to avoid false sharing (line 31). By selecting a suitable array size for `syncVars` (equal to or larger than the number of hardware threads) we ensure that the hardware threads will concurrently access different elements of `syncVars` without unnecessary cache invalidation (line 4).