Scalable Reader-Writer Locks

Yossi Lev
Brown University and
Sun Microsystems Laboratories
yosef.lev@sun.com

Victor Luchangco
Sun Microsystems Laboratories
victor.luchangco@sun.com

Marek Olszewski
Massachusetts Institute of Technology and
Sun Microsystems Laboratories
mareko@csail.mit.edu

ABSTRACT

We present three new reader-writer lock algorithms that scale under high read-only contention. Many previous reader-writer locks suffer significant degradation when many readers attempt to acquire the lock concurrently, even though they are all allowed to hold the lock at the same time. In contrast, our locks scale almost perfectly when there is only read contention on a 4-chip system with a total of 256 hardware threads.

Two of the algorithms extend the MCS queue mutex to provide reader-writer synchronization with low overhead, and can be used when busy-waiting synchronization is appropriate. The third algorithm is an improvement on a production-quality reader-writer lock used in the Solaris™ kernel, which provides robust priority and flexible fairness guarantees.

A key tool we developed to implement our reader-writer locks is the *closable scalable nonzero indicator* (C-SNZI), a variation on the SNZI object. C-SNZI objects allow us to significantly reduce the contention among reads when many readers try to acquire the lock concurrently, but keeps the acquisition overhead small in the absence of read contention. We present an algorithm for C-SNZI that achieves this goal, and show how it can be used by each of our lock algorithms to provide scalable reader-writer locks with different fairness guarantees.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – Parallel Programming; D.4.1 [**Operating Systems**]: Process Management – Synchronization; E.1 [**Data Structures**]: Distributed Data Structures

General Terms

Algorithms, Performance

Keywords

reader-writer lock, scalable, synchronization, SNZI.

1. INTRODUCTION

Reader-writer locks relax the constraints of mutual exclusion (mutex) locks by allowing multiple threads to access a shared object concurrently as long as none of them writes to it, increasing the potential for concurrency when read-only parallelism exists [3]. However, realizing this parallelism is particularly challenging with today's reader-writer locks, which employ serializing updates to central data structures to monitor the number of reader threads acquiring the lock. This serialization can prevent reader-writer locks from scaling linearly with threads, even under read-only workloads. As a consequence, many experts are wary of using reader-writer locks and caution programmers to think twice before using them [2].

While plenty of work has been aimed at improving the performance of reader-writer locks, much of it has focused on reducing interconnect contention caused by busy-waiting on central data structures rather than eliminating the nonscalable serializing accesses to the central data structures. For example, Mellor-Crummey and Scott extended their MCS queue-based mutex lock to support reader-writer exclusion [12, 11]. The MCS lock reduces cache-coherence traffic due to busy-waiting by having each thread spin on a separate and local cache line, with waiting threads forming a queue and each thread notifying its successor in the queue when it releases the lock. In the extension, a reader is allowed to acquire the lock if its predecessor is an active reader, and requires any reader that acquires the lock to notify its successor (if any) if that successor is a waiting reader. In addition, the lock maintains a count of the number of active readers and the next writer in the queue (if any) so that when the last active reader releases the lock, it can notify the writer to acquire the lock. Although this algorithm reduces cache-coherence traffic when the lock is released, every thread still updates the tail pointer when it acquires the lock, and every reader updates the reader count both when it acquires the lock and when it releases it. As a result, this algorithm does not scale well under heavy read contention.

Krieger *et al.* [8] present a lock algorithm (we call it the KSUH lock, following the convention of using the inventors' initials to name the lock) that eliminates the fields containing the reader count and next writer. Instead, it maintains this information implicitly in the queue, which is implemented using a doubly linked list so that a reader releasing the lock can splice itself out of the queue even if its predecessor and successor are still active readers. However, the pointer to the tail of the queue is still updated by every thread, whether reader or writer, and so is still a significant point of contention.

In practice, locks based on busy waiting are used only under special circumstances since they waste CPU resources and are susceptible to priority inversion. As a result, many reader-writer locks found in production code employ heavier-weight strategies that deschedule waiting threads, waking them up according to some policy when the lock becomes available. For example, the Solaris™ kernel uses a reader-writer lock [1] that maintains a queue of waiting threads that have been put to sleep so that they can be woken up when they are at the front of the queue as the lock is released. The queue is protected by a mutex so that threads can be easily ordered according to their priorities and read/write intentions and easily in-

SPAA'09, August 11–13, 2009, Calgary, Alberta, Canada.
ACM 978-1-60558-606-9/09/08.

herit priorities from others. A separate *lockword* is used to guarantee reader-writer exclusion and contains the number of readers that are holding the lock and a flag indicating whether any writer holds the lock (in which case no reader can hold the lock). Like the tail pointer of the above-mentioned locks, this lockword becomes a significant source of unnecessary contention, even under read-only workloads, since it must be updated by every thread every time it acquires and releases the lock.

To eliminate such contention, Hsieh and Weihl [7] propose trading writer throughput for reader throughput by having each thread maintain a private mutex. Under this approach, a reader need only acquire its private mutex to acquire the lock, while a writer must acquire all of them. A similar approach is taken by Dice and Shavit [4] for specialized locks used in their transactional memory system. While this technique provides scalability for read-only workloads, it is feasible only for low numbers of threads as the burden placed on writers becomes excessive at large thread counts.

In this paper, we present three new reader-writer lock algorithms, collectively called OLL locks (again after the inventors' initials), which eliminate updates to central shared data when acquired for reading. To the best of our knowledge, these locks are the first to scale almost perfectly to hundreds of threads under heavy read contention, without trading the performance of write acquisitions. The key observation behind these algorithms is that we do not need to know the number of active readers holding the lock at any time, but only whether there are *any* such readers. A similar observation in different contexts led to the development of *scalable nonzero indicator* (SNZI) objects, which allow threads to "arrive" and "depart" and to determine whether there is a "surplus" of arrivals (i.e., there have been more arrivals than departures). To support reader-writer synchronization, we augment the SNZI interface to allow a SNZI object to be "closed" so that no more arrivals may occur—corresponding to a writer blocking readers—until the object is reopened. The resulting object, which we call a *closable* SNZI (C-SNZI), is the key that allows each of the OLL locks to scale almost perfectly under read-only contention. We describe C-SNZI in detail in Section 2.

The three OLL locks provide different features and different performance characteristics in the presence of writers. The first OLL lock, called the General OLL (or GOLL) lock, is similar to the Solaris kernel lock in that it allows a sophisticated queuing policy by protecting the queue of waiting threads by a mutex lock. The other two OLL locks follow Mellor-Crummey and Scott, and Krieger *et al.*, in extending MCS-style mutex queue locks to allow reader-writer synchronization, but with restrictions on the queuing policy. One implements a "fair" FIFO policy (FOLL lock), while the other gives readers preference by allowing them to advance in front of writers to join other waiting readers in the queue (ROLL lock). We describe these locks in more detail in Sections 3 and 4. Finally, we compare these algorithms with each other and with previous reader-writer locks in Section 5 and conclude in Section 6.

2. CLOSABLE SNZI

A key tool in our reader-writer lock algorithms is a new abstract data type we call the *closable scalable nonzero indicator* (C-SNZI, pronounced "see snazzy"). As its name suggests, C-SNZI is a variant of SNZI [5]. A SNZI object supports three operations, Arrive, Depart and Query, such that Query returns whether there is a *surplus* of arrivals (i.e., whether there have been more Arrive than Depart operations). Depart must not be invoked when there is no surplus. A basic C-SNZI, specified in Figure 1, adds two operations, Close and Open. Arrivals that occur while the C-SNZI is closed "fail": such Arrive operations return false and do not increase the surplus. A Depart operation always succeeds (i.e., it always decrements the surplus), and it returns true unless it decrements the surplus to zero and the C-SNZI is closed (i.e., it is the "last departure" from a closed C-SNZI). A Query operation returns both whether there is a surplus and whether the C-SNZI is open. A C-SNZI is initially open and Open may be invoked only when the C-SNZI is closed and there is no surplus. Close may be invoked at any time; it returns true if and only if the C-SNZI was open and its surplus was (and remains) zero. Because arrivals on a closed C-SNZI fail, once a closed C-SNZI has no surplus, its surplus remains zero until it is opened.

```
shared variables:
    surplus: integer, initially 0
    state: {OPEN, CLOSED}, initially OPEN

Open()
    requires state = CLOSED ∧ surplus = 0
    state ← OPEN

Close(): boolean
    if state = OPEN then
        state ← CLOSED
        return (surplus = 0)
    else
        return false

Arrive(): boolean
    if state = OPEN then
        surplus ← surplus + 1
        return true
    else
        return false

Depart(): boolean
    requires surplus > 0
    surplus ← surplus − 1
    return ¬(surplus = 0 ∧ state = CLOSED)

Query(): (boolean, boolean)
    return (surplus > 0, state = OPEN)
```

Figure 1: C-SNZI specification.

As we show in this paper, C-SNZI is useful for implementing reader-writer locks: to acquire and release the lock, readers use Arrive and Depart respectively, whereas writers use Close and Open.

2.1 Variations

To implement the GOLL lock, we found it useful to extend the C-SNZI interface slightly to provide variants to Open and Close. The OpenWithArrivals operation opens a C-SNZI and simultaneously increases the surplus to an amount specified by a parameter to the operation. It also takes a boolean flag that specifies whether the C-SNZI should then be closed immediately afterward. Thus, an OpenWithArrivals operation is equivalent to atomically executing Open, and then the specified number of Arrive operations, and then, if specified, the Close operation. The CloseIfEmpty operation is exactly like the Close operation except that it does not close the C-SNZI if it has a surplus.

2.2 Implementation

C-SNZI is easy to implement with a counter and a boolean flag that can be updated atomically: the counter maintains the surplus and the boolean flag indicates whether the C-SNZI is open or closed. However, in such an implementation, all Depart and non-failed Arrive operations conflict with each other, and with every Query operation, inhibiting scalability. Such contention is unnecessary when

Arrive and Depart operations change the surplus from one nonzero value to another.

The basic idea for implementing SNZI is to avoid much of this contention by constructing a rooted tree of SNZI objects, where each child is implemented using its parent [5, 9]: Arrive and Depart operations on a child may invoke Arrive and/or Depart on its parent, but only when the surplus at the child might change from zero to nonzero, or vice versa, and in such a way that *the root has a surplus if and only if some node in the tree has a surplus*. This property is also maintained for every subtree of the tree.

With such a tree, threads may arrive and depart at any node of the tree ("corresponding" arrivals and departures should occur on the same node), and propagate upward as necessary to satisfy the property above. Queries can be made directly at the root. Because we require only that the surplus at the root of a subtree is nonzero if and only if the surplus of any of the nodes in the subtree is nonzero, a node with nonzero surplus need not propagate arrivals and departures to its parent, except for the "first arrival" and "last departure", which creates and eliminates the surplus.

The choice of the node to arrive at should be guided by the contention on the SNZI object. When many threads are arriving and departing concurrently, we can reduce contention by arriving and departing at the leaves of the tree. In the absence of contention, arriving and departing at the leaves is expensive because threads must traverse and modify every node on the path from the leaf to the root, so we arrive and depart directly at the root. In our implementation, we adopt the simple policy of arriving at the root unless attempting to do so has failed several times, or if there is already some surplus due to arrivals at leaves. Note that we can avoid allocating the tree (other than the root node) until it is needed, thus incurring the associated space overhead only for those SNZI objects that are heavily contended. Also, rather than remembering the node that a thread arrives at within the SNZI object, it is convenient to return a pointer to it from the Arrive operation, and pass this pointer when invoking the Depart operation. (A failed Arrive operation returns a null pointer.) This pointer should not be dereferenced or manipulated outside the C-SNZI code, and to emphasize this, we encapsulate it in a "ticket" data type.

Figure 2 presents pseudocode for the resulting C-SNZI algorithm. The implementation is based on the algorithm of Lev *et al.* [9], which is simpler than the original SNZI algorithm [5], and has the important property that an Arrive operation that invokes Arrive on the parent does not modify the child node before doing so. Therefore, if an Arrive operation on the parent fails due to a closed C-SNZI, no "cleanup" of the child is necessary. This property enables us to extend the algorithm to support the Open and Close functionality by simply adding a single bit to the root node indicating whether the C-SNZI object is open or closed. This extension is mostly straightforward, with Depart, Query and most Arrive operations linearized at the same points as in the underlying SNZI algorithm, and the Open and Close operations linearized when the new bit is modified. The only subtlety is that when a thread arrives at a leaf of the tree, it may increment the surplus without accessing the root if the surplus is already nonzero. In this case, it cannot atomically determine that the C-SNZI object is open when it increments the surplus. However, the thread checks that the C-SNZI object is open before deciding to arrive at the leaf, so in this case, we can linearize the Arrive operation to the point at which the thread sees that the C-SNZI is open.

Lev *et al.* describe an additional optimization that is required to reduce the contention on the root node of the SNZI tree that can occur when multiple threads attempt to arrive at the SNZI when the surplus is zero. For simplicity, we omit this optimization from the pseudocode, but we use it in the implementation. This optimization does not add any additional CompareAndSwap operations to any of the C-SNZI operations.

3. GENERAL READER-WRITER LOCK

In this section, we introduce a general OLL reader-writer lock (GOLL lock) that is modeled after the Solaris reader-writer lock, which we first describe. Like the Solaris lock, it can be easily modified to provide different fairness guarantees.

3.1 Solaris Lock

The Solaris kernel lock enforces reader-writer exclusion using a single central *lockword* and uses the Solaris turnstile mechanism to queue up threads waiting on a contended lock. Turnstiles are mutex-protected priority queues that control the order with which threads are allowed to acquire a resource. The lockword is composed of a count of active readers, a writeLocked bit, a writeWanted bit, and a hasWaiters bit. The lock is acquired for reading when the reader count is nonzero, and acquired for writing when the writeLocked bit is set. In the absence of conflicting lock requests (concurrent read/write or write/write lock requests), threads simply CompareAndSwap this lockword directly to acquire the lock by incrementing the reader count or setting the writeLocked bit.

In the presence of conflicting lock requests, a thread wanting to acquire the lock sets the hasWaiters bit (and the writeWanted bit if it is a writer), and enqueues itself into the lock's turnstile. Setting the bits and enqueuing the thread must appear atomic so that any thread put to sleep in the turnstile is guaranteed to be woken up by a releasing thread. Thus, the thread acquires the turnstile mutex before performing a CompareAndSwap operation on the lockword to set the appropriate bits, and releases the mutex and restarts if the CompareAndSwap fails.

When no threads are waiting, a thread releases the lock by using CompareAndSwap to decrement the reader count or reset the writeLocked bit as appropriate. If a thread is waiting on the lock (i.e., hasWaiters is set), the last thread about to release the lock does not release the lock, and instead hands over ownership of the lock to the thread(s) next in line to acquire it. In this way, there is no window of opportunity for threads to acquire the lock after it has been released but before the next-in-line thread(s) is/are able to acquire it. A thread handing over ownership to a writer simply sets the writeLocked bit and wakes up the writer. When handing over the lock to a group of waiting readers, the releasing thread sets the reader counter to the number of readers in that group and wakes them up. Thus, threads always own the lock upon awakening.

3.2 Design

Figure 3 presents the pseudocode for the GOLL lock. Rather than using a central lockword, the GOLL lock uses a C-SNZI object to track readers and writers in a scalable manner in the absence of conflicting lock requests. In the presence of conflicting lock requests, the lock orders waiting threads using a central mutex-protected queue. Thus, the form of the GOLL lock follows that of the Solaris lock, except for a few changes required to keep our C-SNZI abstraction. Like the lockword in the Solaris lock, the state of lock is determined by the state of the C-SNZI object. The lock is free if the C-SNZI object is open with no surplus. The lock is acquired for writing if the C-SNZI is closed and has no surplus. If the surplus is nonzero, then the lock is acquired for reading whether the C-SNZI is open or closed. The latter case signifies that a writer is waiting on the lock.

A writer can acquire a free lock by calling CloseIfEmpty on the C-SNZI object, which will write-lock the lock if the C-SNZI is

```
type QueryReturn = record  // Type returned by Query
  nonzero : boolean
  open    : boolean

type Ticket = record  // Type returned by Arrive
  node    : *SnziNode/*SnziRootNode

type CSNZI = record
  root    : SnziRootNode  // Root node in C-SNZI tree
  leafs[] : SnziNode  // Array of leaf nodes in C-SNZI tree

type SnziRootNode = record  // Single CASable word
  count   : int
  state   : enum {OPEN, CLOSED}

type SnziNode = record
  cnt     : int  // Initially 0
  parent  : *SnziNode/*SnziRootNode  // Immutable pointer

// Checks whether the C-SNZI state is OPEN, and if so,
// increments the surplus of the C-SNZI by either directly
// arriving at the root node, or calling TreeArrive on one
// of the leaf nodes. Returns a ticket pointing to the node
// that was arrived at. If the state is CLOSED, makes no
// change and returns a ticket that contains no pointer.
procedure Arrive(csnzi: *CSNZI): Ticket
  while true
    old := csnzi->root
    if old.state != OPEN then return Ticket(null)
    if !ShouldArriveAtTree() then
      new := old
      new.count++
      if CAS(&csnzi->root, old, new) then
        return Ticket(&csnzi->root)
    else
      leaf := &csnzi->leafs[GetLeafForThread()]
      if TreeArrive(leaf) then
        return Ticket(leaf)
      else
        return Ticket(null)
end

// Decrements the C-SNZI surplus. Returns false iff the
// resulting state is CLOSED and the surplus is zero.
// Ticket must have been returned by an arrival. Must have
// received this ticket from Arrive more times than Depart
// has been called with the ticket. (Thus, the surplus
// must be greater than zero.)
procedure Depart(csnzi: *CSNZI, ticket: Ticket): boolean
  return TreeDepart(ticket->node)
end

// Increments the C-SNZI surplus and returns true if the
// C-SNZI is open or has a surplus. Calls TreeArrive
// recursively on the node's parent if needed.
// Otherwise, returns false without making any changes.
procedure TreeArrive(node: *SnziNode): boolean
  arrivedAtParent := false
  repeat
    x := node->cnt
    if x == 0 and !arrivedAtParent then
      if TreeArrive(node->parent) then
        arrivedAtParent := true
      else
        return false
  until CAS(&node->cnt, x, x + 1)
  if arrivedAtParent and x != 0 then
    TreeDepart(node->parent)
  return true
end

// Decrements the C-SNZI surplus, calling TreeDepart
// recursively on the node's parent if needed. Returns
// false iff the resulting state of the C-SNZI is CLOSED
// and the surplus is zero. Otherwise, returns true.
procedure TreeDepart(node: *SnziNode): boolean
  repeat
    x := node->cnt
  until CAS(&node->cnt, x, x - 1)
  if x == 1 then
    return TreeDepart(node->parent)
  else
    return true
end

// Base case for TreeArrive, when we reach the root node.
procedure TreeArrive(root: *SnziRootNode): boolean
  repeat
    old := *root
    if old == SnziRootNode(0, CLOSED) then return false
    new := SnziRootNode(old.count + 1, old.state)
  until CAS(root, old, new)
  return true
end

// Base case for TreeDepart, when we reach the root node.
procedure TreeDepart(root: *SnziRootNode): boolean
  repeat
    old := *root
    new := SnziRootNode(old.count - 1, old.state)
  until CAS(root, old, new)
  return new != SnziRootNode(0, CLOSED)
end

// Opens a C-SNZI object. Requires C-SNZI state to be
// CLOSED and the surplus to be zero.
procedure Open(csnzi: *CSNZI)
  csnzi->root := SnziRootNode(0, OPEN)
end

// Opens a C-SNZI object while atomically performing cnt
// arrivals. Requires C-SNZI state to be CLOSED and
// the surplus to be zero.
procedure OpenWithArrivals(csnzi: *CSNZI, cnt: int,
                           close: boolean)
  if close then
    csnzi->root := SnziRootNode(cnt, CLOSED)
  else
    csnzi->root := SnziRootNode(cnt, OPEN)
end

// Closes a C-SNZI object. Returns true iff the C-SNZI
// state changed from OPEN to CLOSED and the surplus is
// zero.
procedure Close(csnzi: *CSNZI): boolean
  repeat
    old := csnzi->root
    if old.state != OPEN then return false
    new := SnziRootNode(old.count, CLOSED)
  until CAS(&csnzi->root, old, new)
  return new == SnziRootNode(0, CLOSED)
end

// Closes a C-SNZI if its surplus is zero. Otherwise, does
// nothing. Returns true iff C-SNZI state changed from
// OPEN to CLOSED.
procedure CloseIfEmpty(csnzi: *CSNZI): boolean
  repeat
    old := csnzi->root
    if old != SnziRootNode(0, OPEN) then return false
    new := SnziRootNode(0, CLOSED)
  until CAS(&csnzi->root, old, new)
  return true
end

// Returns whether the C-SNZI has a nonzero surplus and
// whether the C-SNZI is open.
procedure Query(csnzi: *CSNZI): QueryReturn
  root := csnzi->root
  return QueryReturn(root.count > 0, root.state == OPEN)
end

// Returns whether the Arrive operation that returned
// the ticket succeeded.
procedure Arrived(t: Ticket): boolean
  return t.node != null
end

// Constructs and returns a ticket that can be used to
// depart from the root node.
procedure DirectTicket(csnzi: *CSZNI): Ticket
  return Ticket(&csnzi->root)
end
```

Figure 2: C-SNZI pseudocode.

```
type RWlock = record
  csnzi    : *CSNZI
  queue    : WaitQueue
  metalock : Mutex

type Local = record
  ticket   : Ticket

type Waiters = record
  kind     : enum {READER, WRITER}
  count    : int
  spin     : bool
  cv       : ConditionVariable
  lock     : Mutex

procedure WriterLock(lock : *RWlock)
  if CloseIfEmpty(lock->csnzi) then return
  Lock(lock->metalock)
  if Close(lock->csnzi) then
    Unlock(lock->metalock)
  else
    waiter := Enqueue(lock->queue, WRITER)
    Unlock(lock->metalock)
    Wait(waiter)
end

procedure WriterUnlock(lock : *RWlock)
  Lock(lock->metalock)
  waiters := Dequeue(lock->queue)
  if waiters == null then
    Open(lock->csnzi)
    Unlock(lock->metalock)
  else
    if waiters->kind == READER then
      OpenWithArrivals(lock->csnzi, waiters->count,
                       lock->queue.numWriters != 0)
    Unlock(lock->metalock)
    Signal(waiters)
end

procedure ReaderLock(lock : *RWlock, local : *Local)
  while true
    local->ticket := Arrive(lock->csnzi)
    if Arrived(local->ticket) then return
    Lock(lock->metalock)
    if Query(lock->csnzi).open then
      Unlock(lock->metalock)
      continue;
    waiter := Enqueue(lock->queue, READER)
    Unlock(lock->metalock)
    // Thread releasing the lock will pre-arrive directly
    // for us.
    local->ticket := DirectTicket(&lock->csnzi)
    Wait(waiter)
    return
end

procedure ReaderUnlock(lock : *RWlock, local : *Local)
  if Depart(lock->csnzi, local->ticket) then return
  Lock(lock->metalock)
  waiters := Dequeue(lock->queue)
  // Policy may let readers overtake a waiting writer,
  // which closed the C-SNZI.  So if readers are next,
  // re-open the C-SNZI directly into the CLOSED state.
  if waiters->kind := READER then
    OpenWithArrivals(lock->csnzi, waiters->count, true)
  Unlock(lock->metalock)
  Signal(waiters)
end
```

Figure 3: GOLL lock pseudocode.

open and has no surplus (i.e., if the lock is free). If the lock is already acquired for either reading or writing, the writer's call to CloseIfEmpty will fail. In this case, the writer atomically closes the C-SNZI object and inserts itself into the wait queue by acquiring the queue mutex and then calling the Close operation on the C-SNZI object. If Close returns true (i.e., the C-SNZI was open with no surplus), then the lock was no longer held, and the Close operation has successfully acquired it for writing. Thus, the writer simply releases the mutex and returns. Otherwise (i.e., Close returns false), the writer enqueues itself in the wait queue and goes to sleep (by waiting on a waiter object that uses a condition variable to put the thread to sleep), knowing that one or more threads have not yet released the lock, and that one of them will wake the writer up when it releases the lock (recall that the Close operation returns true if and only if the C-SNZI is open and with a surplus of zero. i.e., if the lock is free).

As with the Solaris lock, a writer releasing the lock hands over ownership of the lock to a waiting writer or group of readers, if any exists. To do so, a releasing writer acquires the mutex to check whether any threads are waiting on the lock. If not, the lock is released by simply opening the C-SNZI object and releasing the mutex. If a writer thread is next in line, it is sufficient to release the mutex and wake it up to complete the hand-over, since the lock is already in a write-locked state. If the lock is to be handed over to a group of readers, the writer converts the lock to the read-acquired state by performing an OpenWithArrivals operation on the C-SNZI, which sets the C-SNZI surplus to the number of waiting readers, and, if there are no writers waiting on the lock, changes its state to open. Next it releases the mutex and wakes up the readers.

In the absence of conflicting lock requests, readers can acquire the lock by performing an Arrive operation on the C-SNZI object, which returns a ticket that specifies the node at which the arrival took place. Since arrivals always succeed when the C-SNZI object is open, the mutex is never accessed for read-only workloads. If the C-SNZI is closed, either because the lock is write-locked or because it is read-locked but with a writer waiting, the Arrive operation will fail and the reader must enqueue itself into the mutex-protected queue. To guarantee that it will be woken up after it goes to sleep, a reader ensures that the C-SNZI remains in a closed state after it has acquired the queue mutex. Finally, the reader creates a ticket for itself which indicates that it should depart from the root node. Recall that a writer releasing the lock performs an OpenWithArrivals operation on the root node on behalf of any waiting readers.

A reader releasing the lock starts by calling the Depart operation on the C-SNZI, passing the ticket it received when acquiring the lock. If no threads are waiting on the lock (i.e., the C-SNZI is open) or the reader is not the last to depart the C-SNZI, the Depart operation returns true and the reader is done. Otherwise (i.e., Depart returns false), the C-SNZI is closed and the thread was the last to depart. Thus, the reader must hand the lock over to the waiting thread. The reader first acquires the mutex to dequeue the next thread(s) waiting to acquire the lock. If a writer is next in line, the reader can simply release the mutex and wake it up, since the final Depart operation already placed the lock into a write-locked state (closed C-SNZI with no surplus). Otherwise, the lock is to be handed over to a group of readers,[1] so the reader hands the lock over by calling OpenWithArrivals to place the lock back into a read-locked state.

3.2.1 Supporting Write Upgrade

Many production reader-writer locks, including the Solaris lock, support a *write-upgrade* operation: a thread that holds the lock for reading can upgrade its read-ownership to write-ownership if it is the only thread holding the lock (if not, the write-upgrade operation fails and the thread keeps holding the lock for reading). Unfortunately, although checking whether a reader is the sole thread holding the lock is trivial when using a counter that tracks the number of readers holding the lock, doing so using a C-SNZI object is less

[1]The scheduling policy employed in the wait queue may have chosen to let readers overtake a waiting writer due to thread priorities.

easy since the C-SNZI object indicates only whether some readers are holding the lock, and not how many.

We can add write-upgrade support to the GOLL lock by splitting the counter at the root node into two counters: one that is updated by the Arrive and Depart operations that are invoked directly on the root (a direct counter), and a second that is updated by the Arrive and Depart operations that are invoked on the root by operations on its children. Using these counters, a thread can check whether it is the only reader holding the lock as follows:

- If the thread acquired the lock for reading by arriving directly at the root, then it is the only thread holding the lock if and only if the direct counter is one, and the other counter is zero.
- Otherwise, the thread executes an Arrive operation on the root node, and then a Depart operation on the node it has originally arrived at. After this point, the thread is the only one holding lock if and only if the direct counter is one, and the other counter is zero.

In other words, because the direct counter shows the *exact* surplus of Arrive operations invoked directly on the root, the thread simply "trades" its arrival at the tree with a direct arrival at the root, and then checks the counters to see whether the surplus of direct arrivals is greater than one, or whether the surplus of arrivals at nonroot nodes is nonzero.

4. DISTRIBUTED QUEUE LOCKS

In this section, we present two distributed queue-based OLL reader-writer locks that offer higher performance with simpler fairness criteria. Like the reader-writer locks by Mellor-Crummey and Scott [11] and by Krieger *et al.* [8], we extend the MCS mutex lock [10] to support reader-writer synchronization. Like those algorithms, our new algorithms improve performance when a lock is contended by reducing the number of writes to central shared data that a thread needs to perform to enqueue itself into a wait queue. Unlike those algorithms, however, successive readers in our algorithms do not use separate nodes in the queue. Instead, they use C-SNZI to share a single node in the queue. We first provide background on the MCS mutex lock algorithm, then present our FOLL algorithm, which guarantees first-in first-out (FIFO) fairness, and finally describe how to modify it to achieve the ROLL lock, which guarantees reader-preference FIFO fairness.

4.1 The MCS Mutex Lock

The idea behind the MCS mutex lock [10] is to maintain an implicit queue of nodes belonging to waiting threads. Every thread has its own node, which it enqueues every time it wants to acquire the lock. Each node has a spin flag and a qNext pointer, and the lock consists of a single pointer pointing to the tail of the implicit queue of waiters (null if the queue is empty).

To acquire the lock, a thread uses a FetchAndStore operation to store its node, with spin $=$ false and qNext $=$ null, to the tail of the queue. If the previous tail (returned by the FetchAndStore operation) was null, then the thread immediately enters its critical section. Otherwise, it sets its spin flag to true, changes the previous tail's qNext pointer to point to its node, and waits until its spin flag is false, after which it enters its critical section.

A thread leaving the critical section checks if it has a successor (i.e., its node's qNext pointer is not null). If so, then it sets its successor's spin flag to false. Otherwise, it uses CompareAndSwap to change the tail from its node to null. If this succeeds, then no thread is waiting for the lock, so it is done. If the CompareAndSwap fails, then some other thread put a pointer to its node into the tail, so the exiting thread waits until its qNext pointer is updated by the other thread, at which point it sets its successor's spin flag to false.

4.2 FOLL Reader-Writer Lock

As mentioned above, the FOLL lock uses a C-SNZI object to allow successive readers to share a single node. Such readers, after the first, avoid writing the tail pointer, and instead simply arrive at and depart from the C-SNZI object. Writers close the C-SNZI object and insert their own node to form a queue as in the MCS mutex lock. A reader following a writer also adds a node to the queue, but readers immediately following it can join its node using the C-SNZI object of that node. Thus, read-only workloads avoid writing the tail pointer entirely, eliminating a major source of contention on the lock.

In the FOLL lock, we have two kinds of nodes, one for readers and the other for writers. Writer nodes are essentially the same as the nodes of the MCS mutex lock; each thread has its own writer node. Reader nodes have three extra fields, one of which points to a C-SNZI object. The other two fields are used to recycle reader nodes as discussed at the end of this section.

A writing thread acquires and releases the lock in almost exactly the same way as it would with an MCS mutex lock. The only difference is that if its predecessor is a reader node, then after setting its predecessor's qNext pointer, it closes its predecessor's C-SNZI object, preventing other readers from arriving at that C-SNZI. If the Close operation returns true, which means that the C-SNZI is empty (i.e., has no surplus) when it is closed, then there are no readers to signal the writer (i.e., set its spin flag to false), so the writer spins instead on the spin flag of its predecessor. Also, due to the node recycling algorithm we discuss later, the C-SNZI object of reader node may not be open when a writer attempts to close it, in which case the writer simply waits until the C-SNZI is opened before proceeding as described above.

When a reader wants to acquire the lock, it first examines the tail. If it points to a reader node, then the thread simply attempts to arrive at the C-SNZI of that node. If it succeeds, then it waits for the spin flag of that node to be false (it may already be so), and then enters the critical section. If it fails to arrive, then some writer must have closed the C-SNZI after enqueuing behind that node (that is the only operation that closes C-SNZI objects), so the tail must have changed, and we simply retry the operation.

If tail is null then the reader gets an unused reader node, which, when just allocated, has a closed C-SNZI with no surplus and qNext $=$ null. The reader sets the node's spin flag to false, and attempts to enqueue the node using CompareAndSwap to change the tail from null to point to the node. If it succeeds, then it opens the C-SNZI of the node it just enqueued, and then we are back to the previous case, where the tail is a reader node. If it fails, the tail pointer must have changed, indicating that some other thread must have enqueued another node, thus again, we retry the operation.

Similarly, if the tail is a writer node, the reader tries to enqueue an unused reader node after the writer node. However, in this case, it first sets the spin flag of the node to be enqueued to true (as the reader must wait for the writer to release the lock), and after it enqueues the node, it changes the predecessor's qNext pointer to point to the newly enqueued node.

Note that we do not open the C-SNZI object of a node until the node has been enqueued. This is important to prevent readers from arriving at a C-SNZI of a node that is not in the queue, which could otherwise happen because of node recycling, as discussed below. In FOLL, a C-SNZI is opened only immediately after it is enqueued by a reader, and it is not removed from the queue and recycled until it is closed (by a writer that enqueues behind it) and has no surplus.

```
type RWlock = record
   tail          : *Node
   rNodes        : *Node  // Head of reader node

type Node = record
   kind          : enum {READER, WRITER} // Immutable
   qNext         : *Node
   spin          : boolean
   // The following fields are used only by READER nodes
   csnzi         : *CSNZI
   allocState    : enum {FREE, IN_USE}
   next          : *Node

type Local = record
   rNode         : *Node  // Default read node. Immutable
   wNode         : *Node  // Write node.  Immutable.
   departFrom    : *Node  // List node we last arrived at.
   ticket        : Ticket // C-SNZI ticket

// Allocates a new reader node.
procedure AllocReaderNode(local : *Local)
  currNode := local->rNode
  while true
    if currNode->allocState == FREE then
      if CAS(&currNode->allocState, FREE, IN_USE) then
        return currNode
    currNode := currNode->next
end

// Frees a reader node.  Requires that its allocState
// is IN_USE.
procedure FreeReaderNode(N : *Node)
  N->allocState := FREE
end

procedure WriterLock(lock : *RWlock, local : *Local)
  oldTail := FetchAndStore(&lock->tail, local->wNode)
  if oldTail != null then
    local->wNode->spin := true
    oldTail->qNext := local->wNode
    if oldTail->kind == WRITER then
      repeat until !(local->wNode->spin)
    else
      // Wait until node is properly recycled
      repeat until Query(oldTail->csnzi).open
      // Close C-SNZI of previous reader node.
      // If there are no readers to signal us, spin on
      // previous node and free it before entering
      // critical section.
      if Close(oldTail->csnzi) then
        repeat until !(oldTail->spin)
        FreeReaderNode(oldTail)
      else
        repeat until !(local->wNode->spin)
end

procedure WriterUnlock(lock : *RWlock, local : *Local)
  if local->wNode->qNext == null then
    if CAS(&lock->tail, local->wNode, null) then
      return
    else
      repeat until local->wNode->qNext != null
  local->wNode->qNext->spin := false
  local->wNode->qNext := null          // Clean up
end
```

```
procedure ReaderLock(lock : *RWlock, local : *Local)
  rNode := null

  while true
    tail := lock->tail

    // If no nodes are in the queue
    if tail == null then
      if rNode == null then
        rNode := AllocReaderNode(local)
      rNode->spin := false
      if CAS(&lock->tail, null, rNode) then
        Open(rNode->csnzi)
        local->ticket := Arrive(rNode->csnzi)
        if Arrived(local->ticket) then
          local->departFrom := rNode
          return
        rNode := null  // Avoid reusing inserted node

    // Otherwise, there is a node in the queue
    else
      // Is last node a writer node?
      if tail->kind == WRITER then
        if rNode == null then
          rNode := AllocReaderNode(local)
        rNode->spin := true
        if CAS(&lock->tail, tail, rNode) then
          tail->qNext := rNode

          local->ticket := Arrive(rNode->csnzi)
          if Arrived(local->ticket) then
            local->departFrom := rNode
            repeat until !(rNode->spin)
            return
          rNode := null  // Avoid reusing inserted node

      // Otherwise, last node is a reader node.
      // (tail->kind == READER)
      else
        local->ticket := Arrive(tail->csnzi)
        if Arrived(local->ticket) then
          if rNode != null then FreeReaderNode(rNode)
          local->departFrom := tail
          repeat until !(tail->spin)
          return
end

procedure ReaderUnlock(lock : *RWlock, local : *Local)
  if Depart(local->departFrom->csnzi, local->ticket) then
    return
  local->departFrom->qNext->spin := false
  local->departFrom->qNext := null  // Clean up
  FreeReaderNode(local->departFrom)
end
```

Figure 4: FOLL lock pseudocode.

A reader leaving its critical section simply departs from the C-SNZI at which it arrived. If it is the last to depart from a closed C-SNZI (i.e., if the Depart operation returns false), then it also signals its successor (the writer that closed the C-SNZI) by setting its spin flag to false, and recycles the reader node so that it can be reused.

Pseudocode for the FOLL lock is given in Figure 4.

4.2.1 Node Recycling

In the MCS mutex algorithm and the MCS and KSUH reader-writer extensions, each thread always uses exactly one queue node, which it enqueues when trying to acquire the lock, whether for reading or for writing, and dequeues when releasing the lock. This is also true for the writer nodes of the FOLL lock. However, it is *not* true for the reader nodes because many readers may share the same node while waiting in the queue (or executing their critical section). The thread that enqueued the node may not be the last to depart from it; moreover, the last thread to depart from the node may do so after the thread that enqueued that node wants to acquire the lock for reading again, and thus requires another reader node.

Therefore, the FOLL lock algorithm maintains a pool of reader nodes to be used by threads that acquire the lock for reading. We provide two auxiliary procedures to manage this pool of reader

nodes: AllocReaderNode returns a reader node that is not being used, and FreeReaderNode takes a node that was returned by AllocReaderNode and makes it available for future calls of AllocReaderNode. AllocReaderNode is called by ReaderLock when it needs to enqueue a new node onto the lock queue. FreeReaderNode is called when the node is removed from the queue, either by the last reader that departs from a closed C-SNZI (which signals the writer that closed the C-SNZI to enter its critical section) as part of ReaderUnlock, or by a writer that closes the C-SNZI of a node that has no readers when it is closed as part of WriterLock. It is also called by a ReaderLock operation that allocates a new node that it never actually puts in the queue (because some other reader added a reader node first).

It is thus easy to see that an allocated node is never freed twice before it is reallocated: either it is never placed on the queue (and therefore its C-SNZI is never opened since it was allocated), in which case it is freed by the thread that allocated it but did not insert it into the queue, or it is placed in the queue and its C-SNZI is opened, in which case it is freed when it is removed from the queue by the thread that made its C-SNZI both closed and with no surplus (i.e., the thread that invoked either Depart or Close and got false in return). There is at most one of the latter kind of thread because once a C-SNZI is closed with no surplus, it remains that way until it is opened again, which is done only by threads that allocate the associated node and insert it into the queue.

It is also easy to see that an allocated reader node will always be freed provided that all operations terminate and that the node does not remain in the queue: a node that is allocated and not put in the queue is freed by the ReaderLock that allocated it, and a node removed from the queue is freed by the operation that removes it.

As mentioned above, it is important that the C-SNZI of a node be closed if it is not in the queue. Otherwise, a reader may see a node at the tail of the queue, and then get delayed before arriving at the node's C-SNZI. Since the thread has yet to modify the state of the C-SNZI, a writer can close the C-SNZI object and the node can be subsequently dequeued and recycled once all other readers depart. If the C-SNZI object were opened before its node is put back into the queue, the sleeping reader may awaken and arrive at the C-SNZI without noticing that the node is not in the queue. Since the node may never be placed back into the queue, the thread may never be able to enter the critical section. Worse yet, if the spin flag of the node is false, the delayed thread may enter the critical section prematurely, possibly while a writer has the lock, violating reader-writer exclusion.

To implement the pool, we use a ring of nodes, with each node containing a pointer to the next node in the ring and an allocState field indicating whether the node is free or in use. To reduce contention, we assign each thread a distinct default node, from which it starts to traverse the ring (following next pointers) looking for a free node. We then use CompareAndSwap to atomically change the node from a free node to an in-use node. To free a node, we simply change its allocState field to indicate it is free; we need not use CompareAndSwap because, as argued above, at most one thread will attempt to free it until it is reallocated.

Finally, we show that whenever a thread requests a reader node (i.e., calls AllocReaderNode), there is one available. Because each thread has a distinct default node, there are N reader nodes that may be allocated, where N is the number of threads. We show that this is sufficient by associating a distinct thread with each node in use (i.e., not free) at any time. For a reader node that has been removed from the queue but is not yet free, we associate the thread that removed it, which must still be executing ReaderUnlock (a writer that removes a reader node from the queue frees it immediately). For a node that has been allocated but not yet since been inserted into the queue, we associate the thread that allocated it (which must still be executing ReaderLock). For a reader node in the queue but not at the tail, we associate the writer that owns the immediately following node (which must be a writer node because reader nodes cannot be adjacent in the queue). Finally, for a reader node at the end of the queue, we associate the thread that allocated it, which cannot be any of the aforementioned threads because that node has been at the tail of the queue since the thread inserted it (so the thread will not request another reader node).

4.3 ROLL Reader-Writer Lock

When strict FIFO ordering is not required, the FOLL Lock can be improved by allowing readers to overtake waiting writers to join a collection of waiting readers that have yet to acquire the lock. We call such a lock a *reader-preference OLL lock*, or ROLL lock.

A key requirement of the ROLL lock is the ability to find a reader node in the queue. In the FOLL lock, as in the MCS locks, nodes have pointers to the next node in the queue, but threads cannot traverse back up the queue. We make the FOLL lock into a ROLL lock by converting the wait queue into a doubly linked list. When a reader attempts to acquire the lock, it traverses from the tail towards the head in search of a reader node. If it finds such a node, then it checks the spin flag to see if the readers using that node are still waiting to acquire the lock (i.e., if spin $=$ true). Because all readers follow this procedure, there can be at most one such reader node. If the flag is true, the reader arrives at the C-SNZI of that node, joining the other readers waiting at that node and begins busy-waiting on the spin flag. If the spin flag is already false, or if no such node is found, then the thread creates a new reader node and enqueues it at the end of the queue, as in the FOLL lock.

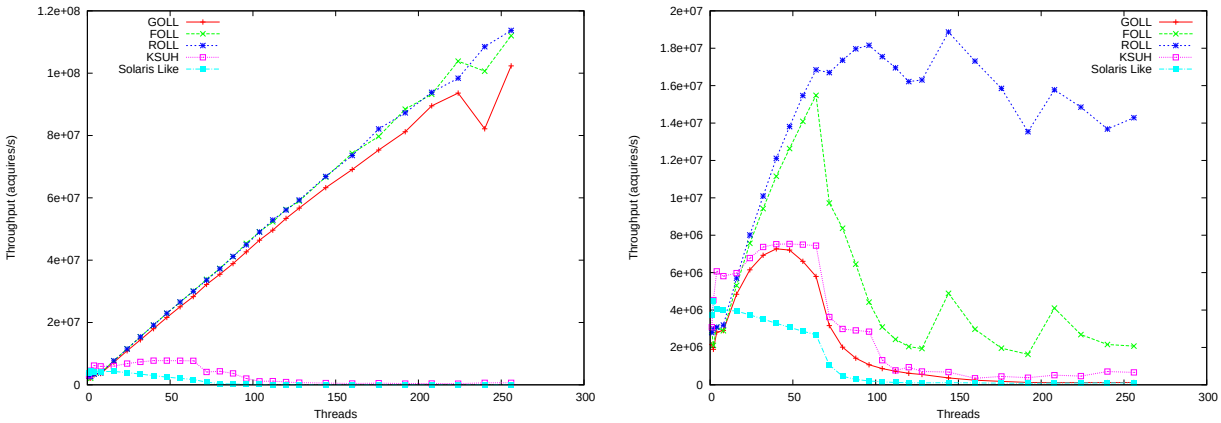
As an optimization, we also maintain in the lock object a pointer to the last known reader node with threads still busy-waiting. The pointer is updated whenever a thread finds such a node, and is set to null whenever a thread fails to join the node. The optimization reduces the number of searches that need to be performed to find the last reader node.

5. EVALUATION

In this section, we present experimental results for each of our lock algorithms, and compare them against previous reader-writer lock algorithms.

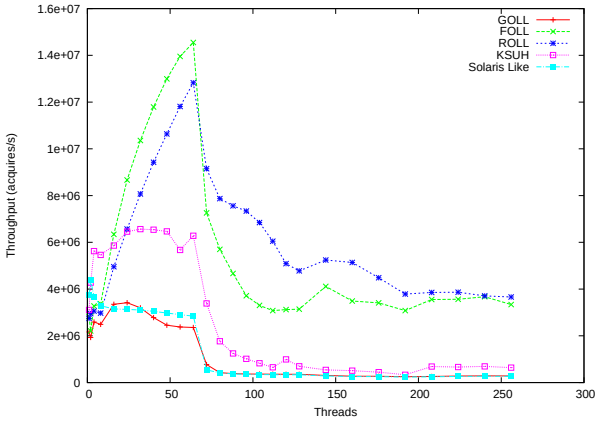
5.1 Methodology

In addition to our lock algorithms, we implemented versions of the KSUH lock and the Solaris kernel lock (the Solaris implementation cannot be used in user-space). The KSUH lock was the fastest MCS-style reader-writer lock we found. To ensure a fair comparison, we tuned the exponential back-offs for each lock independently. For the GOLL lock, we used the same fairness policy used by the Solaris lock: readers hand the lock over to writers, and writers hand the lock over to readers (unless a higher-priority writer is waiting). For both the GOLL and Solaris-like locks, we used our own spin-based condition variables to eliminate the cost of context switching. In addition, we found that the locks performed best across all contention levels if we maintained two counters at the root of the C-SNZI tree (instead of the one described in Section 2): one for arrivals that have propagated up from the tree and one for direct arrivals to the root node. This allows the ShouldArriveAtTree function called during the Arrive operation to favor direct arrivals until it encounters contention *or* until it sees that other threads have arrived using the tree, indicating that contention was recently observed by another thread.

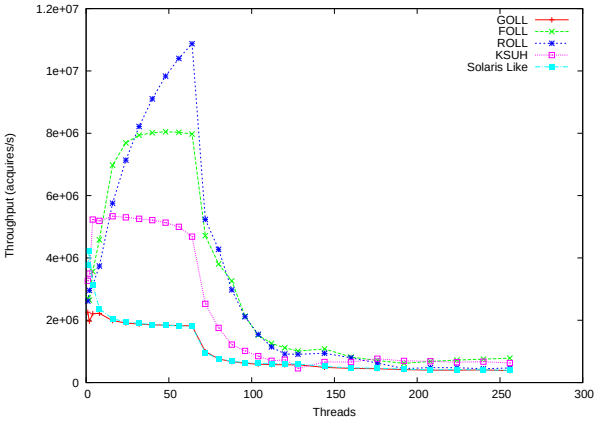


(a) 100% Reads

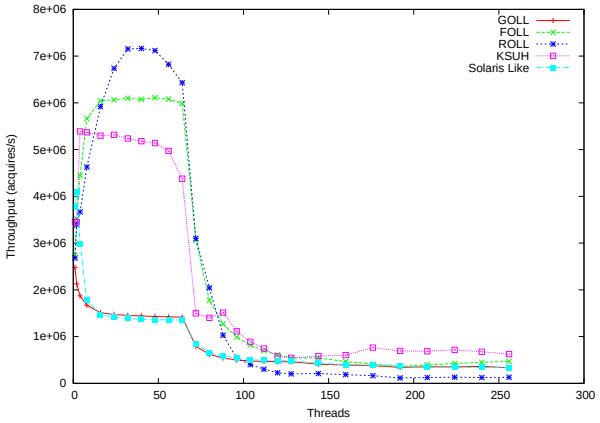
(b) 99% Reads



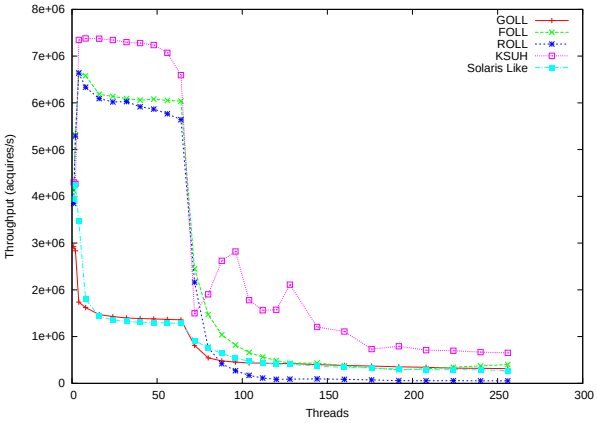
(c) 95% Reads



(d) 80% Reads



(e) 50% Reads



(f) 0% Reads

Figure 5: Throughput results for reader-writer locks at various levels of read/write contention.

We evaluated the performance of each lock by making threads repeatedly acquire and release the lock in a tight loop without performing any work within the critical section. Threads decide whether to acquire the lock for reading or writing using a per-thread private random number generator and a target read percentage. We obtain the average throughput of each lock by measuring the amount of time needed for all threads to complete 100,000 lock acquisitions (for read percentages of 50% or less, we did only 10,000 lock acquisitions, to compensate for the longer run times caused by the mutual exclusion of the writers). All threads were assigned the same priority. We ran each experiment three times and present the average of the results.

We performed our experiments on a 1.4 GHz Sun SPARC Enterprise® T5440 server [6], which combines chip-level multithreading (*CMT*) and symmetric multiprocessing (*SMP*) designs. The T5440 server contains four UltraSPARC® T2 Plus processors con-

nected via four UltraSPARC T2 Plus XBR coherency hubs. Each T2 Plus processor contains a single 4 MB L2 cache shared by eight cores, each of which supports 8 hardware threads, for a total of 64 hardware threads per chip. Thus, the machine supports up to 256 hardware threads, but inter-thread communication overhead increases significantly when running more than 64 threads, at which point not all threads can communicate via a shared L2 cache.

5.2 Experimental Results

Figure 5 presents the throughput results for each of the locks across a wide range of thread counts and for various levels of read-/write contention. Figure 5(a) shows the performance of the locks for a purely read-only workload. Under this scenario, the throughput of the Solaris-like lock gradually decreases as more threads are added. The KSUH lock is able to offer slight performance improvements up until 64 threads, after which the high cost of communication causes a 10x drop in performance. In contrast, all the OLL locks scale linearly as more threads are added, unaffected by the change in communication cost at 64 threads. At 256 threads, the locks perform two orders of magnitude better than the KSUH lock.

Figure 5(b) displays the performance of the locks when 99% of the lock acquisitions are by readers and 1% by writers. Under this workload, the GOLL lock suffers significantly compared with its read-only performance, scaling slowly up to 48 threads, after which the contention on the queue mutex causes its performance to drop. Nevertheless, the GOLL lock performs significantly better than the Solaris-like lock for which throughput decreases steadily after 2 threads. The KSUH lock performs marginally better than the GOLL lock, exhibiting a similar drop in performance at 64 threads. On the other hand, both the FOLL and ROLL locks continue to scale linearly while communication remains on-chip, and outperform the KSUH lock all the way to 256 threads. With the FOLL lock, beyond 64 threads, the increased cost of communication exacerbates the cost of the serialization due to the mutual exclusion required by the writers, causing a dramatic performance drop of almost 10x. However, by relaxing the FIFO guarantee offered by the FOLL lock, the ROLL lock maintains almost all of its 64-thread performance even with the high cost of off-chip communication.

At a 95% read rate (Figure 5(c)), both the ROLL and FOLL locks continue to scale while executing on a single chip. In contrast, the KSUH lock fails to scale, even while on-chip. At 64 threads, it performs more than 2x slower than the ROLL and FOLL locks. Beyond 64 threads, as the cost of communication goes up, both the FOLL and the ROLL locks exhibit a similar drop in performance. Despite this drop, both locks perform over 5x faster than the KSUH lock at 256 threads. At this read/write ratio, and for all subsequent read/write ratios, the GOLL lock behaves almost exactly like the Solaris-like lock because the cost of acquiring and releasing the queue's mutex dominates.

At an 80% read rate (Figure 5(d)), the ROLL lock continues to scale while on-chip, while the performance of the FOLL lock levels off at 32 threads. Once off-chip, both locks begin to exhibit performance similar to the remaining locks.

At 50% and 0% read rates (Figures 5(e) and 5(f)), all the distributed queue-based locks begin to behave the same: the locks maintain close to constant on-chip and off-chip throughputs with a large drop at 64 threads. Likewise, the GOLL and Solaris-like locks are able to maintain constant throughputs when the cost of communication is the same, though with lower throughputs than achieved by the distributed queue locks when executing on a single chip.

6. CONCLUSION

We presented three new reader-writer locks that provide scalability to hundreds of threads running on processors in a multichip system. A key component used by our locks is the new C-SNZI data structure, which significantly reduces contention when many readers attempt to acquire a lock concurrently, while keeping both space and time overhead low when there is no contention.

We showed how to implement C-SNZI, and how it is used to implement our scalable reader-writer locks. One of these locks improve a production-quality reader-writer lock used in the Solaris kernel, which provides a flexible mechanism for implementing robust priority and fairness policies. The other two locks are queue-based locks that are appropriate for situations permitting busy-waiting synchronization. We demonstrate that these locks scale almost perfectly in the presence of read-only contention among 256 hardware threads running on a 4-chip multiprocessor.

7. REFERENCES

[1] Source code for kernel reader-writer lock in Open Solaris. Open Solaris OpenGrok, 2009. [online] `http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/common/os/rwlock.c`.

[2] B. Cantrill and J. Bonwick. Real-world concurrency. *Communications of the ACM*, 51(11):34–39, 2008.

[3] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, 1971.

[4] D. Dice and N. Shavit. TLRW: Return of the read-write lock. In *TRANSACT '09: Proceedings of the Fourth ACM SIGPLAN Workshop on Transactional Computing*, 2009.

[5] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable nonzero indicators. In *PODC '07: Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 13–22, 2007.

[6] G. Grohoski. Niagara-2: A highly threaded server-on-a-chip. In *HotChips 18: Proceedings of the Eighteenth IEEE HotChips Symposium on High-Performance Chips*, 2006.

[7] W. C. Hsieh and W. E. Weihl. Scalable reader-writer locks for parallel systems. In *IPPS '92: Proceedings of the Sixth International Parallel Processing Symposium*, 1992.

[8] O. Krieger, M. Stumm, R. Unrau, and J. Hanna. A fair fast scalable reader-writer lock. In *ICPP '93: Proceedings of the 1993 International Conference on Parallel Processing*, pages 201–204, 1993.

[9] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *TRANSACT '09: Proceedings of the Fourth ACM SIGPLAN Workshop on Transactional Computing*, 2009.

[10] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[11] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *PPoPP '91: Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 106–113, 1991.

[12] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *ASPLOS-IV: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 269–278, 1991.