**Implementing the Readers-Writer Lock**

Stephen TwiggUniversity of California Berkeley, CS267 Final Project

**Abstract**

*The most basic mutex lock considered for parallel programs allows only one node access to some data at any one time. The readers-writer lock loosens these restrictions by allowing multiple nodes parallel read access while still allowing nodes to get exclusive access for write operations. This paper reviews previous work in readers-writer locks and examines methods for improving speed at which readers are able to begin their parallel reads after a writer releases data while keeping needed network communication reasonably bounded. Modifications to the classic queue structure are proposed to aggregate readers into single objects on the queue. The new structure is then used to allow application programmers to customize the policy they use to wakeup pending readers. It was found a binary tree wakeup structure provides the best scaling whereas naïve spinlocks gave the best performance when the system was tested on the NERSC Hopper cluster using UPC.*

**1. Introduction**

Parallel applications using a shared memory programming model require many synchronization constructs such as locks, barriers, etc. to ensure data be kept consistent and different process communicate properly. A readers-writer lock is a type of synchronization construct that allows either multiple parallel readers or a singular writer on a piece of data at any one time. The priority between allowing readers versus allowing writers is highly dependent on the desired application.

Many applications use constructs that can be considered variants of the readers-writer lock. Valid flags, for example, on data form a common occurrence. Readers spinlock on the valid flag until the owner/writer releases the data by setting valid. Unfortunately, naïve implementations of this variant have weak performance switching from the readable to writable (invalid) state and spinlocks can result in highly inefficient communication. Database applications allowing parallel servicing of read-only queries while still permitting occasion modification of that data also form another example of this lock. Many implementations of cache coherence protocols in hardware can be decomposed as a collection of readers-writer locks held on each cache line. Those schemes employing MOESI cache states are designed to allow either multiple nodes to share read-only access to the object or one node to have writeable access at any one time.

**2. Related Work**

Mellor-Crummey and Scott [1] present multiple implementations of the readers-writer lock. The simple, fair implementation is a very common classic employing a request and completion counters. An agent, upon desiring the lock, ‘generates a ticket’ by atomically incrementing the request counter corresponding to their requests type (read or write) and storing the values of both counter types. A reading agent then spinlocks until the writer completions counter is equal to the value of the write requests when its ticket was generated. A writer agent must spinlock until both reader and writer completions are equal to the value of the read and write requests, respectively, when its ticket was generated. Once a writer or reader finishes with the data, they must atomically increment the appropriate completion counter. As Mellor-Crummey and Scott admit, the principle drawback of this lock is that the waiting agents must continuously spinlock on remote data, the completion counters, which can put strain on the network.

Mellor-Crummey and Scott present an alternative structure for the readers-writer lock that avoids spinlocking on remote data by using queues. The fair version has all agents append the address of some local sentinel to a linked-list queue. Upon activation, the agent removes itself from the queue immediately. Reader agents must increment an active readers counter and wakeup the head of the queue if it is a reader. Upon completion, writer agents immediately activate the next element in the queue. Reading agents must decrement the counter and, if zero, wakeup the next element in the queue (which is almost always a writer). Priority can be granted to writers or readers by generating two separate queues for readers or writers. Only when the prioritized queue is empty are agents in the other queue allowed to activate.

McKenney [2] presents a ‘distributed’ implementation of the readers-writer lock. For each readers-writer lock, a set of standard mutex locks are generated such that there is one for each possible agent in the system. A reader must only secure its mutex lock in order to perform a read operation whereas a writer must secure all mutex locks in the set. This simple system is tempting for its simplicity. Reader lock acquisition latency is very fast and can be made to spinlock on local data with appropriate allocations. Unfortunately, writer lock acquisition latency and memory requirements scale linearly with the number of agents in the system. Also, writers are likely to spinlock on remote data and the system does not readily allow for adjusting the priority of readers and writers. Hsieh [3] presents an adjustment to the system wherein active readers add themselves to a list for the lock. Writers are then required to only wait for locks in that list thus reducing latency slightly. However, this correction still does not address writer remote spinlock and prioritization issues.

**3. Motivation**

Previous work in readers-writer locks provides varying methods to achieve the desired functionality of the lock. Most methods even permit enforcing fair priority between readers and writers or preferring one over the other, dependent on the application. However, the previous methods have not done any significant optimizations of the performance, particularly acquisition latency and memory requirements. The latency between a writer completing and readers beginning data use is of particular interest since this metric is what differentiates the readers-writer lock from other locks. The other transitions of a writer or the last reader completing and then waking up a writer are already well-studied. In those cases, a single agent must wakeup another single agent, a situation already handled by literature for standard mutex locks.

The expected contention of the lock will affect the ideal implementation for the lock. Locks covering data required by only a couple of readers at a time likely does not require elaborate scalable wakeup logic compared to locks covering data desired by many readers at a time. The presence of cache coherence protocols in systems further complicates analysis of readers-writer lock implementations by limiting the penalty of remote spinlocking. These combined factors motivate the development of a highly parameterizable readers-writer lock implementation. This would permit the use of autotuning of lock parameters to the system hardware and parallel application.

Section 4 overviews the proposed implementation for readers-writer locks and considers theoretical performance. Section 5 presents experimental methodology and results for evaluating the writer to reader transition schemes. Section 6 concludes and examines areas of future work.

**4. Motivation**

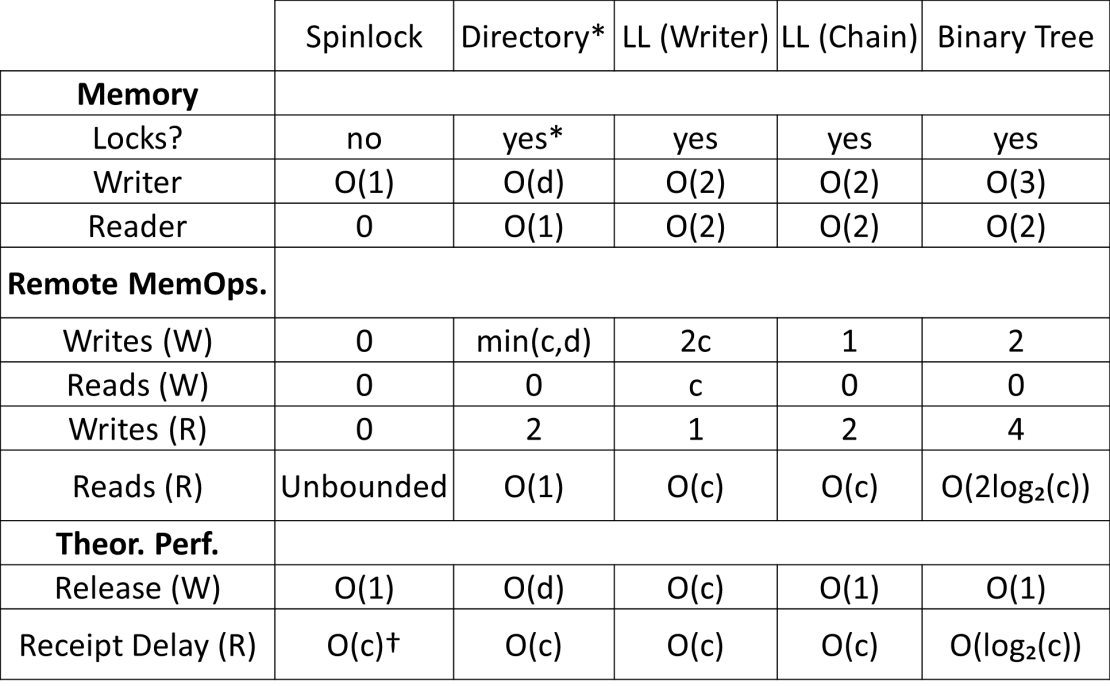
Performance of the writer to readers transition is highly contingent on the reading contention for the data. However, the readers to writer and writer to writer transitions are somewhat independent of that contention and where prioritization of readers versus writers is most significant. Thus, the proposed implementation attempts to provide orthogonality between how the different transition types are handled. This allows for finer tuning of lock performance to the application.

**4.1 Handling the Writer to Readers Transition**

The naïve mechanism for one writer to activate multiple parallel readers is to use spinlocking. The writer owns a local valid signal, which it flags as true once it has finished using the data. Awaiting readers spinlocking on that value can then proceed immediately. This requires the minimal memory amongst all schemes: only the writer must keep one semaphore indicating the data is readable. Similar, the writing process does not have to engage in no remote memory transactions when releasing the lock. Unfortunately, as previously discussed, readers must perform an unbounded number of network reads (dependent on time the writer spends before releasing the data) before receiving the lock. Due to limitations in the hardware/software stack, the memory controller handling the semaphore can only service a finite number of readers at a time. Average delay between writer release and reader receipt is thus expected to scale according to the number of potential readers into the system.

All other structures attempt to keep remote communication bounded by requiring each reader to allocate a local sentinel value. Readers then must simply insert themselves into some data structure. Swapping data structures allows autotuning to optimize the performance under different hardware and expected contention. In general, this will necessitate the use of a mutex lock to ensure readers insert themselves into the data structure atomically. Chart 1 summarizes all theoretical results for these transitions.

**4.1.1 Directory Scheme**

The first structure considered is inspired by the use of directories in cache coherence protocols. The writer for the lock preallocates a contiguous array for holding a set number of pointers to reader sentinels. Readers simply append themselves to the directory. Upon release, the writer alerts the sentinel in each directory entry. If the directory is full, overflowing readers are forced to spinlock and their performance degrades accordingly. Thus, the directory length would ideally be tuned to be slightly larger than the expected number of readers for the lock. Regarding performance, if a reader count is maintained and the directory is not full, readers must only make one remote read of the reader count and then a two remote writes to increment the count and insert a pointer to their sentinel into the directory. The writers must spend deconstructing the directory is bounded and proportional to the directory size. Since the writer must iterate through the list sequentially, a readers receipt delay is proportional to its position on the list.

**4.1.2 Linked List**

Chart 1 – Comparison of Writer to Readers transition methods. d = max. directory size, c = # of current pending readers, W = Writer, R = Reader. Non-unary scalars in O notation included to better emphasize relative differences.  
\* Entries that overflow directory revert to spinlock behavior. These entries do not need to use the lock.  
† O(c) due to hardware/software backend is expected; however, O(1) likely seen due to hardware speed

The next structure considered is inspired by the previous work in readers-writer locks that used queues. The writer for the lock holds merely a single pointer to the first element of the reader queue. Each element holds a pointer to some sentinel and the next element on the list. Each reader is responsible for allocating space for the next element in the list to prevent needed memory in the writer from scaling as the number of readers increases. Also, since each reading process is responsible for allocating memory to service its own request, this prevents agents in the system from starving others of memory systems due to this structure. The chart assumes only the head of the pointer is stored, thus requiring the linked list be traversed to append to the tail. At the cost of more memory space and another write, the number of remote reads done by pending readers can be reduced to O(2) by also storing the tail of the list.

Alerting the readers the data is available after the writer releases can be done using two separate methods. The first calls for the writer to traverse the linked list, alerting readers in succession. This requires the writer to perform sequential remote reads to collect the linked list data and perform all the remote writes necessary to alert the readers’ sentinels. As long as a reader’s sentinel is not signaled until the next element in the list is read, readers can be trusted to clean up the linked list elements from their own memories. This method shares the same scaling of writer release time and reader receipt time scaling with a non-overflowing directory structure. However, it is expected for this to perform slightly slower due to the need to perform remote reads in traversing the linked list.

The alternative method is to chain reader alerts, similar to the method employed in the Muller-Crummey and Scott queue lock. The writer now must only alert the first reader in the linked list. Upon alert, this reader then alerts the next in the chain, and so on. This has the benefit of bounding the number of remote operations the writer must perform (only 1, alert the first reader) and time the writer spends releasing. Scaling of reader receipt time similarly does not change. Absolute performance is likely to improve since less remote reads must be done before a given reader is awakened. The principle disadvantage of this method is that it effectively requires readers to be actively spinlocking on their sentinel. Readers working on other tasks and only occasionally checking the sentinel would delay propagation of the wake signal to readers further down the chain.

**4.1.3 Binary Tree**

The final structure proposed for handling the readers to writer transition is inspired by common methods for handling collective operations. Broadly, waking up the collection of readers is roughly identical to permitting processes to progress past a barrier once the last process has reached the barrier. Previous research has found binary trees to be beneficial for this task [4]. Wakeup must be done by chaining through the readers to see any benefit (as the writer traversing the tree would unhelpfully result in sequential alerts) and thus this system carries the caveat readers must actively spinlock on their sentinel to prevent propagation delay. Thus, the writer must only hold pointers to the sentinels of two other readers. Each reader similarly must have space to hold pointers to two other sentinels. As with the linked list, the needed memory for this structure is bounded for the writer and scales for readers according to the number of pending read requests (which is likely to be at most one in many applications). Writer release time is again bounded by a constant value. Maximum reader alert receipt time; however, now scales by the base-2 logarithm of the number of pending readers (versus linearly for all other methods). Unfortunately, while a tail pointer cannot be maintained to allow for very fast insertions, only log base-2 of the number of pending readers remote reads must be done to traverse the tree to find the shallowest available sentinel slot.

The binary tree must be kept roughly balanced to prevent performance from degrading to that of the linked list. Keeping a count of the number of pending readers allows for a small trick to make this very easy. Reading the binary representation of the count (read before incrementing to account for the current addition) from the least significant bit to the most significant bit provides a traversal route to the shallowest empty spot in the tree.

**4.2 Potential Impact of Cache Coherence Protocols**

The previous analysis of writer to readers transitions ignored the impact of a cache coherency protocol on performance, particularly when spinlocking on remote data. A protocol has the potential to vastly reduce the quantity of network messages generated when the number of agents is less than. In fact, as per the design, the cache coherence protocol transforms the communication behavior of naïve spinlocking into that of the directory-based scheme. While reader response latency will still increase linearly with directory size, the resolution is done in hardware and thus, for low to moderate contention, is likely to outperform software-based schemes with better scaling. However, for data whose contention is anticipated to be higher than the directory length, a scaling software scheme should be employed to prevent directory thrashing from causing network traffic from again becoming unbounded.

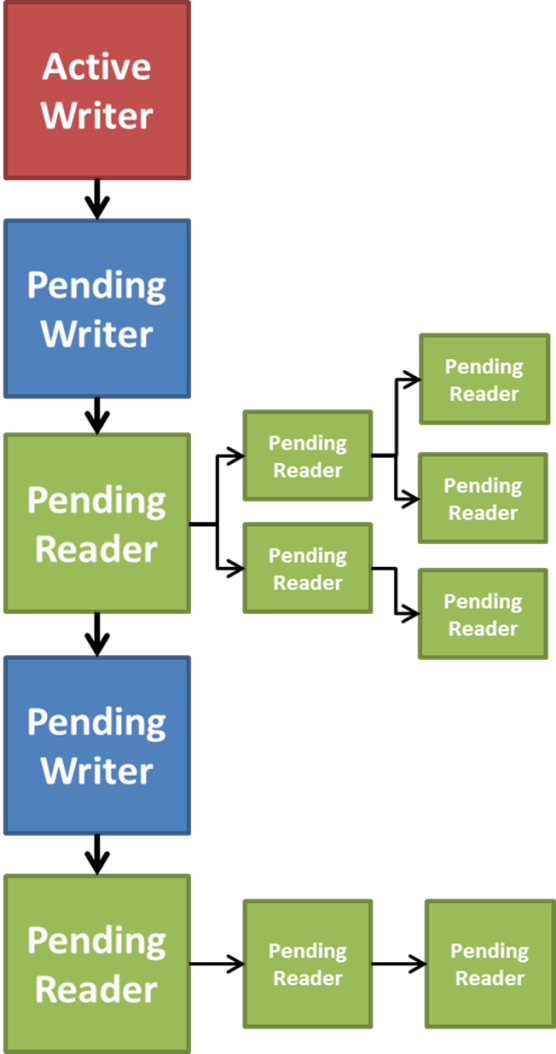
**4.3 Handling Transitions to Writers**

The exact mechanism of executing the readers to writer and writer to writer transitions is highly dependent on how the application designer wishes to prioritize readers versus writers, which may change within an application depending on the data being protected by the lock. For example, a readers-writer lock could be used to guard constantly updating I/O data. A single I/O process, upon receiving new input, would secure the lock for writing, use system calls to gather the input from various registers and memory locations, post the data to a convenient memory location, and finally flag the data as valid to readers. If the input is constantly changing and has the property that old data is unhelpful, the writing I/O process should be able to grab the lock even if readers are in the process of reading data. (An alternative example of this scenario involves a process speculative executing on some input data. If the process must update valid, it is desirable the process grab the lock immediately to update data since any other processes down the chain are executing with garbage anyway.)

The scenarios presented can easily be resolved by using an incrementing ‘age counter’ rather than a flag to indicate whether the data is readable. Writers must set the age to some special invalid value when updating data and then, as part of releasing the lock, set the age to an incremented value. The same structures as discussed in the writer to readers transition section can be used to handle readers trying to access invalid value. Readers are required to compare the age of the data before and after reading the data and restart the attempted read if they are different. This prevents a slow reader from accidentally reading data that was invalid or changed partway through reading.

Applications, however, are likely to require some sort of mechanism for uninterruptible reads to prevent continuous writing from perpetually stalling readers. The classical method for resolving this scenario is to employ an active readers counter. During the writer to readers transition, an active readers counter is atomically set to the number of pending readers in the structure as part of the release process. Readers which enter the system after release must atomically increment the counter and readers which finish their read operations must atomically decrement. Writes are only allowed to initiate when there are no pending active readers. This structure has the unfortunate potential to starve writers as readers enter and leave the system. A possible resolution is, if the data is sufficiently small, to copy the data wholly to readers upon release (or later read) such that an active readers count is unnecessary. However, this solution is untenable for data of appreciable size or when readers are interested in only a small portion of the data that copying the entire record is too wasteful.

**4.4. Combined Solution**

The proposed solution modifies the queue structure with fair priority and bounded communication developed by Mellor-Crummey and Scott. Rather than mixing readers and writers into the same queue, the queue instead is composed only of writers and ‘reader aggregate objects.’ This system acts identically to when these queues are used for standard mutex locks wherein only one agent is permitted to act on the data as a time. The ‘reader aggregate object’ is woken up as if it is a reader and secures exclusive access as if it was performing writes but instead uses its critical section to permit multiple parallel reads to the data.

The reader aggregate object is created by (and thus in the memory space of) the first reader for a block of data after it had been marked as unavailable. This object contains the structure necessary to implement the desired writer to readers wakeup scheme of those outlined in the previous section as well as a pending readers counter. For example, the object may hold simply a valid semaphore that pending readers are instructed to naively spinlock on. Alternatively, it can hold a directory of pointers to reader sentinels (as well as a valid semaphore to allow remote spinlocking if the directory fills up) or a set of pointers needed to implement a tree or linked list reader wakeup structure. All subsequent readers simply must check if a readers aggregate object is located at the tail of the writers queue and append themselves to the structure as discussed in the previous sections. If a fair priority system is in play, a writer appends itself to the writers queue and thus ‘closes out’ a preceding readers aggregate object requiring future readers to generate a new one.

Figure 1 – Combined structure for a readers-writer lock allowing for arbitrary reader wakeup schemes

Upon activation, the first reader, which owns the readers aggregate object, becomes an effective writer and is responsible for resolving the structure to wakeup (or initiate the chaining wakeup) other pending readers. Before doing so, the number of pending readers is copied to the locks number of active readers. Once a structure is resolved, the first reader is free to deallocate the readers aggregate object. Now-activated readers, including the first reader, are required to atomically decrement the active readers counter. If the counter is zero, that reader also must wakeup and transfer ownership of the overall lock to the next object in the queue (which is likely a pending writer). There is a slight potential for slowdown due to the decrement bottleneck; however, this is readily doable in hardware and (unlike for readers wakup which should ideally happen simultaneously) readers are likely to already spend varying amounts of time reading the protected data thus naturally providing spacing.

This structure allows application designers to easily change the wakeup scheme employed for the writer to readers transition not only on a per-application or per-lock basis but also after a set period of time to reflect changes in expected lock contention over time. A summary of this structure is presented in Figure 1. Differing priority schemes are also readily implementable. Readers can be granted priority over writers by allowing readers to append themselves to the first readers aggregate object in the writers queue it encounters, even if other writers present below the object in the queue. Writers can be granted priority by allowing the writer to insert itself in the queue before readers aggregate objects. The queue system allows for even more complex priority schemes by allowing higher priority writers to skip ahead of lower priority writers. Readers can similar be granted a sliding priority amongst writers by simply ensuring the readers aggregate object priority is always the highest of all the priorities of contained readers. Since the reads are all done in parallel, this allows the lower priority readers to take advantage of the presence of a very high priority reader without causing significant priority inversion if medium priority writes are pending.

**5. Experimental Evaluation of Reader Wakeup Schemes**

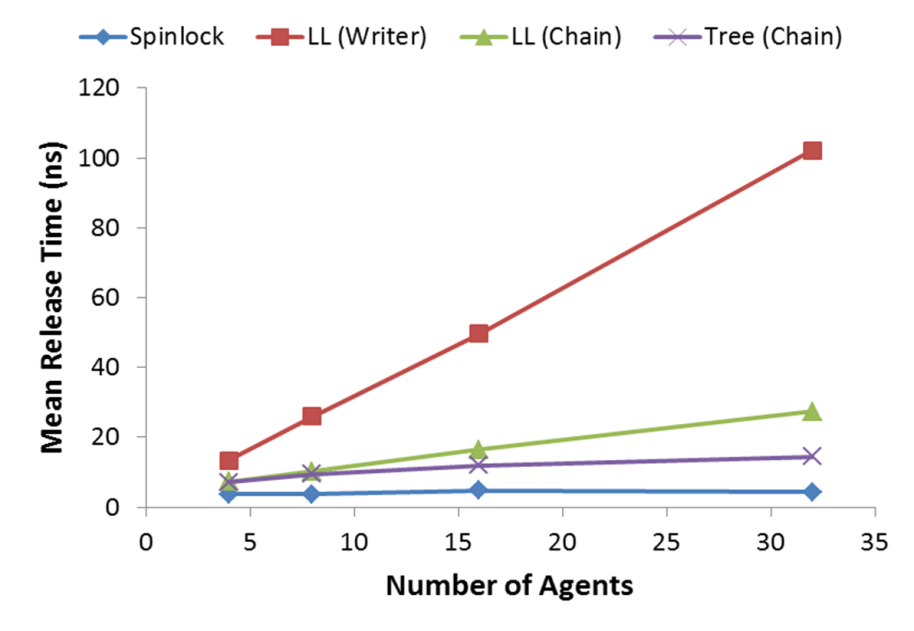
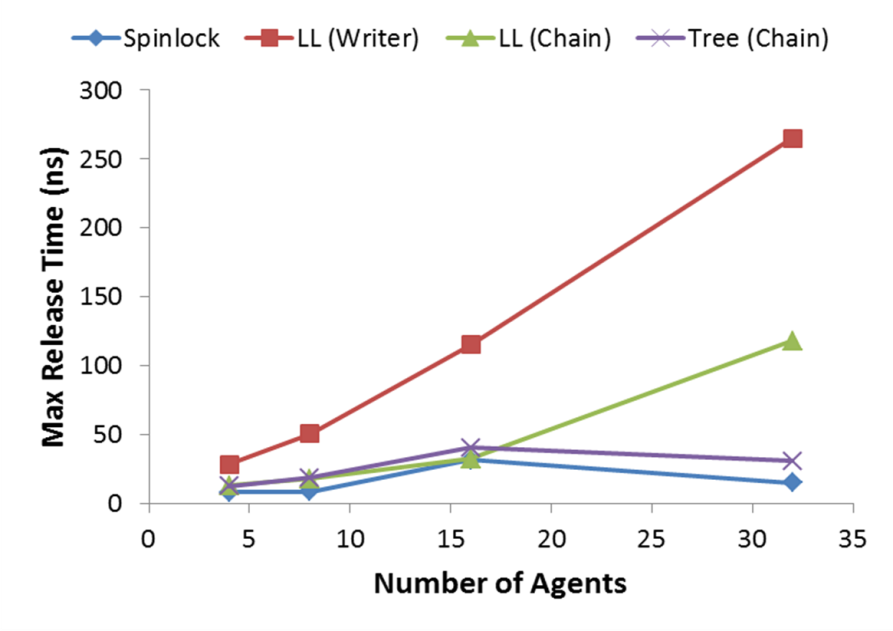
The various schemes for implementing the writer to readers transition were tested using UPC on the NERSC Hopper cluster. The naïve spinlocking, linked list, and binary tree release structures were all implemented and timed using up to 32 agents with 32 trials for each structure, agent count combination. The directory structure was implemented and worked for most trials but plagued by an occasional deadlock issue that ruined select trials and required exclusion of the structure from the chart. In all cases, the readers were assumed to be actively seeking the read and thus available to complete the chain immediately. For each trial, the writer would immediately acquire the lock and unlock after one second. During the locking time, the readers seek the lock data, adding themselves to the wakeup structure and spinlocking on the appropriate local sentinel or remote semaphore.

Figure 2 – Evaluation of various writer to readers transition scheme on the NERSC Hopper cluster using UPC. In each trial, 1 agent acted as the writer while the others setup as parallel pending readers. Each reader was setup to submit their read request before data was released.

Response latencies were calculated by synchronizing process timings with a UPC barrier and comparing the time readers leave their spinlock (indicating data is available) and the writer began processing wakeup requests. Unfortunately, processes are not guaranteed to leave the UPC barrier at the same time thus ensuring perfect time synchronization. However, averaging over multiple trials is expected to correct for this issue and at least expose general scaling trends and relative performance between schemes.

Figure 2 shows both the mean latency amongst all readers for all trials as well as the maximum latency reported by any one reader in any trial. The linked list schemes exhibit strong linear scaling in response time as the number of agents increase as expected. Naïve spinlocking came out ahead with the highest response time; likely because the method exploits spinlock semaphore resolution delay is composed only of hardware delays whereas other schemes incur software-related overhead. Of the schemes minimizing expected remote communication, the binary tree structure was squarely ahead of the other methods and almost matched the speed of naïve spinlocking. The structure also exhibits the logarithmic scaling anticipated.

**6. Conclusions and Future Work**

An extension of previous implementations for the readers-writer lock was proposed. The base structure requires a queue of writers and grouped readers, allowing only one object to resolve at a time. The readers, grouped into a readers aggregate object, are awakened according to the policy selected to minimize reader wakeup latency for the expected contention level. Priority between readers and writers can be manipulated by adjusting where agents are permitted to be inserted into the queue. This customizable structure allows application designers to tailor performance according to the application and system hardware. Evaluation of writer to readers wakeup policies indicated the binary tree yields the best scaling. However, evaluation of the schemes on the NERSC Hopper cluster using UPC shows naïve spinlocking still has the potential to yield the best results due to advantages of a hardware-reliant approach over those requiring software controls.

Future work would evaluate the combined solution as a whole to better optimize the performance of the whole structure.

**7. References**

[1] John M. Mellor-Crummey and Michael L. Scott. 1991. Scalable reader-writer synchronization for shared-memory multiprocessors. SIGPLAN Not. 26, 7 (April 1991), 106-113.

[2] Paul E. McKenney. 1996. Selecting locking primitives for parallel programming. Commun. ACM 39, 10 (October 1996), 75-82.

[3] Wilson C. Hsieh and William E. Weihl. 1992. Scalable Reader-Writer Locks for Parallel Systems. In Proceedings of the 6th International Parallel Processing Symposium (IPPS '92), Viktor K. Prasanna and Larry H. Canter (Eds.). IEEE Computer Society, Washington, DC, USA, 656-659.

[4] Rajesh Nishtala. 2009. Automatically Tuning Collective Communication for One-Sided Programming Models. Ph.D. Dissertation. University of California at Berkeley, Berkeley, CA, USA. Advisor(s) Katherine A. Yelick.