[introduction]

[RELATED WORK]

Mellor-Crummey and Scott [CITATION] 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 [CITATION] 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 [CITATION] 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.

[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 theorizes on the potential impact of cache coherence protocols on those results. Section 7 concludes and examines areas of future work.

[IMPLEMENTATION]

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. Pseudo-code outlining these applications can be found in the Appendix.

[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 REFERENCE] summarizes all theoretical results for these transitions.

[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.

[LINKED LIST]

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.

[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 [NISHTALA THESIS]. 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.

[readers -> writer transition]

[evaluation]

[theoretical chart discussion]

[NERSC latency data discussion]

[writer-> writer, writer -> readers structure]

[readers aggregate object]

[conclusions]

[appendix: implementation pseudocode]