Scalable Reader-Writer Locks for Parallel Systems

by

Wilson C. Hsieh William E. Weihl

November 1991

Abstract

Current algorithms for reader-writer synchronization exhibit poor scalability because they do not allow readers to acquire locks independently. We describe two new algorithms for reader-writer synchronization that allow parallelism among readers during lock acquisition. We achieve this parallelism by distributing the lock state among different processors, and by trading reader throughput for writer throughput; we expect that in highly concurrent programs, the ratio of writers to readers should be very low, so this tradeoff should lead to better overall performance. We used a multiprocessor simulator, Proteus, to compare these algorithms with existing algorithms. Our experiments show that when reads are a large percentage of lock requests, the throughput of both of our algorithms scales significantly better than current algorithms; even when there is a fair percentage of writes, the throughput of one of our algorithms still scales better than current algorithms.

Keywords: Parallel systems, Reader-writer locking, Synchronization.

This research was supported by the National Science Foundation under grant CCR-8716884, by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988, and by an equipment grant from the Digital Equipment Corporation. The first author was also supported by a National Science Foundation Graduate Fellowship.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

1 Introduction

Synchronization has been an area of research in computer science since the 1960's. Although synchronization mechanisms for uniprocessors are well understood, the same cannot be said for multiprocessors. Two factors increase the relative cost of synchronization on multiprocessors: network communication costs and resource contention. Since synchronization occurs quite frequently on multiprocessors, it is important to develop high-throughput, low-contention synchronization mechanisms. In this paper we describe our research on reader-writer synchronization for shared-memory multiprocessors.

Recent work has focused on developing good algorithms for mutual exclusion on shared-memory multiprocessors [2, 12, 14]. Lamport's work [12] concentrates on minimizing the number of remote memory accesses; unfortunately, his algorithm leads to spinning over the network in the absence of caches. Anderson's lock [2] and Mellor-Crummey and Scott's lock [14] are similar: they both use fetch-and-add (or similar operations) to eliminate remote spinning; as a result, their algorithms generate very little network traffic.

Reader-writer locks relax the constraints of mutual exclusion: a given reader-writer lock can be held by multiple readers, but can only be held by one writer at a time; in addition, a given lock cannot be held simultaneously by both readers and writers. Such a lock can be used to ensure the sequential consistency of shared memory, as long as all readers and writers obey the locking protocol. Reader-writer locks are extremely useful for multiprocessors, since shared memory is a common abstraction on multiprocessors. Our interest in reader-writer locks lies in the fact that there is an opportunity for parallelism among readers that has not been exploited in previous work.

Reader-writer locks were first described by Courtois et al. [8] in 1971. Recent papers by Mellor-Crummey and Scott [14, 15] have presented several new algorithms for reader-writer synchronization in shared-memory multiprocessors. Their algorithms are based on the use of atomic operations such as "fetch-and-add" and "compare-and-swap", and are extensions of their work on algorithms for mutual exclusion synchronization.

Mellor-Crummey and Scott call their algorithms for reader-writer synchronization "scalable"; unfortunately, they define "scalable" to mean that the throughput of lock acquisition remains constant as the number of processors that request the lock increases. This definition of scalability, although satisfactory for mutual exclusion algorithms, is unsatisfactory for reader-writer synchronization. In particular, if all (or a large number of) processes are readers, the throughput of lock acquisition should increase linearly with the number of processors, since the semantics of read-locks does not preclude multiple readers from acquiring locks in parallel.

We have developed two new algorithms for reader-writer synchronization that take advantage of this opportunity for parallelism. Our algorithms provide scalable throughput for readers; they differ in the tradeoffs that they make between the costs for readers and for writers. Using Proteus [4, 5, 9], a multiprocessor simulator, we compared their performance with two standard algorithms. When a large proportion of lock requests are reads, one of our algorithms scales substantially better than recent algorithms. Indeed, our results show that previously described algorithms [15] do not scale at all.

In Section 2 we describe our two new algorithms: *static* and *dynamic* reader-writer locks. In Section 3 we describe the experiments that we performed to compare our algorithms with two other algorithms. Section 4 evaluates the algorithms, Section 5 discusses related and future work, and we draw conclusions in Section 6.

2 Our Algorithms

We present two new spin-waiting algorithms for reader-writer synchronization on shared-memory multiprocessors: the *static* and the *dynamic* algorithms. These algorithms were designed to handle one process per processor; however, they could be easily modified to handle multiple processes per processor. We view network delay and contention as the primary issues in achieving high-performance synchronization; the additional overhead to support multiple processes per processor should not affect the relative performance of our algorithms. Our model assumes that shared memory is partitioned among the processors; each processor has "local" shared memory that can be accessed without going over the network.

All of our algorithms use spin-waiting, but readers in the *static* and *dynamic* algorithms spin only on local memory. We use "semaphore" to mean a word of memory that is accessed using test-and-set with exponential backoff; we do not use test-and-test-and-set, as we do not assume the presence of caches.[1]

The literature describes several different fairness conditions that can be implemented [3, 8, 16]: first-come-first-serve, reader-priority, and fair-readers priority are but a few. We have not implemented any of these conditions, as we consider fairness to be a secondary concern. The addition of fairness constraints would involve additional synchronization, which would probably lead to a reduction in performance.

2.1 Static algorithm

Our *static* algorithm allocates one semaphore per processor, plus an extra semaphore that acts as a gate on writers. In order to acquire a *static* lock, a reader acquires a local semaphore, whereas a writer must acquire all of the semaphores. The gate is present to keep the writers from interfering excessively with readers. In an earlier version that did not have the gate, the performance of the first reader was poor, because all of the writers were trying to acquire its semaphore. Figure 1 outlines the structure of a *static*

[1] We place a limit on the backoff; without a limit, we quickly overflowed 32 bits in most of our experiments.

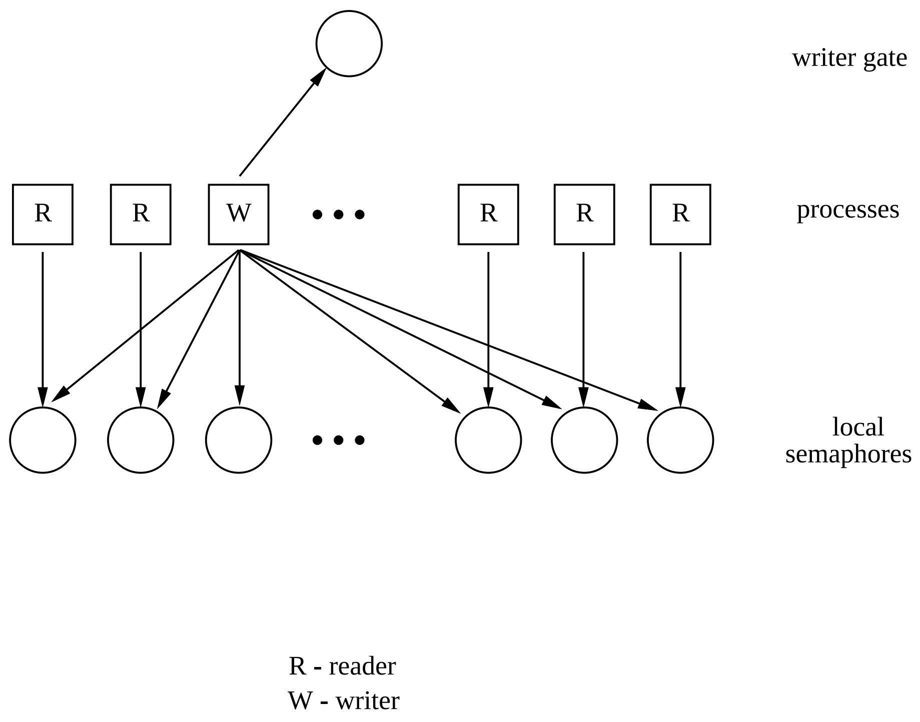

Figure 1: Static reader-writer lock

lock: an arrow from a process points to a semaphore that the process must acquire in order to acquire the reader-writer lock. When releasing a *static* lock, a reader simply releases its local semaphore; a writer releases all of the semaphores. The lock can be viewed as a quorum of semaphores, where the read quorum is 1 and the write quorum is n. We call this algorithm *static* since the quorum is static.

The *static* lock has two important properties: readers do not interfere with each other, and readers do not have to go over the network to acquire a lock. This algorithm performs extremely well when all lock requests are reads: a reader only needs to acquire a local semaphore. Unfortunately, the fact that readers never interfere means that writers must do a substantial amount of work in systems with many processors; when even a few percent of the requests are writes, the throughput suffers dramatically, since a writer must acquire a semaphore on every processor in order to acquire a *static* lock.

2.2 Dynamic algorithm

Our *dynamic* algorithm provides each processor with a semaphore and a bit that indicates whether the semaphore is "valid." Figure 2 outlines the structure of a *dynamic* lock (note that the reader queue and valid list are pointers to shared-memory). A *dynamic* lock can be viewed as a variant of the *static* lock that keeps track of the active readers in a centralized location; this saves work for writers, as they do not

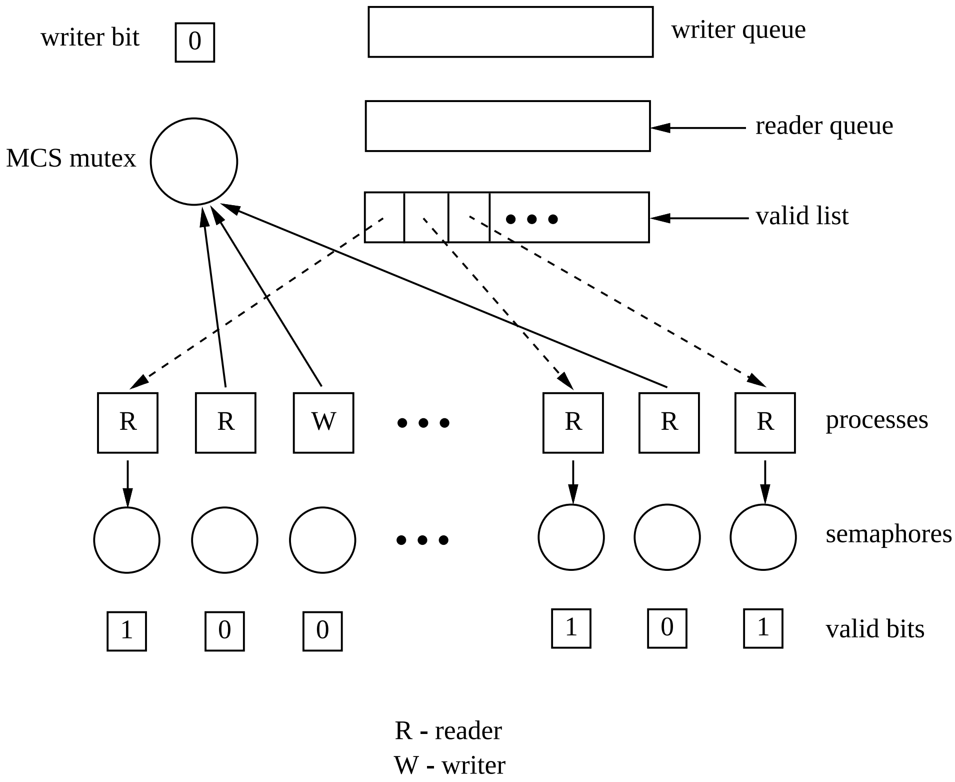

Figure 2: Dynamic reader-writer lock

have to contact every processor on every lock acquisition and release. We call this algorithm *dynamic* since the write quorum is dynamically maintained.

The list of valid semaphores is protected by a Mellor-Crummey and Scott (MCS) mutual exclusion lock [14].[2] If a reader tries to acquire a *dynamic* lock and its semaphore is valid, it can just acquire the semaphore[3]; otherwise, it must acquire the MCS mutex. After the reader has acquired the mutex, it checks the writer bit. If there is an active writer (i.e.. the writer bit is set), it adds itself to the reader queue, releases the mutex, and waits; otherwise, it adds its semaphore to the list, marks its semaphore valid and acquires it, and releases the mutex. Waiting is implemented using semaphores: the waiting process enqueues the address of its local semaphore, sets the semaphore, releases the mutex, and then spins until the semaphore is cleared; the process that performs the wakeup just clears the semaphore. In order for a reader to release a *dynamic* lock, it just needs to release its local semaphore.

In order for a writer to acquire a *dynamic* lock, it must first acquire the mutex protecting the list. If

[2] The MCS mutual exclusion lock should not be confused with their reader-writer lock, although they are very similar.

[3] However, the reader must check the validity of its semaphore after acquiring the semaphore; if it has become invalid, the reader must acquire the mutex.

there is an active writer (i.e., the writer bit is set), it puts itself on the writer queue, releases the mutex, and waits. Otherwise, it sets the writer bit and releases the mutex. The writer then invalidates all of the semaphores on the valid list, clears the valid list, and then waits for each semaphore to be cleared, which ensures that there are no readers with locks. The writer then has the lock, and can continue. A writer unlocks a *dynamic* lock by first reacquiring the mutex, and then checking if there is a waiting writer. If a writer is waiting, the unlocking writer wakes it up, and then releases the mutex. If there is no waiting writer, it checks if there are waiting readers. If there are, it swaps the pointers for the valid list and the reader queue, makes all of the waiting readers' semaphores valid, clears the writer bit, and releases the mutex; if there are no waiting readers, it just clears the writer bit and releases the mutex.

Although readers can interfere with each other in this algorithm, they do not always do so; two readers can only interfere with each other when they are both trying to acquire the MCS mutex to make their semaphores valid. The reason that we make them interfere is to reduce the amount of work that a writer must do; this decreases the throughput of readers, but increases the throughput of writers and the overall throughput. In addition, although a reader might have to go over the network to acquire a lock, it does not have to go over the network if its local semaphore is valid.

3 Experiments

We compared the performance of various algorithms for reader-writer synchronization using the Proteus simulator, a high-performance, reconfigurable multiprocessor simulator that was developed at MIT [4, 5, 9]. Proteus is an execution-driven simulator that interleaves the execution of an application program with the simulation of the underlying architecture. This structure makes it possible for Proteus to achieve a high degree of accuracy, which has been confirmed by several experiments [7, 9]. We configured Proteus to run our experiments on k-ary 1-cubes, 2-cubes, and 3-cubes of various sizes: 1, 2, 4, 8, 27, 64, and 125 processors. Although the Proteus simulator does allow us to simulate shared-memory caches, we did not do so, as we did not want the choice of caching scheme to affect our results. In addition, we wanted to make the movement of data as explicit as possible.

We measured throughput under various different percentages of read requests: 0%, 50%, 75%, 95%, 99%, and 100%. In each experiment, every process executes a tight loop; at each iteration, each process chooses whether to act as a reader with percentage p.[4] When any process completes 400 iterations, the experiment stops. When $p = 99$, however, the experiment runs until 1600 iterations complete; the larger number of iterations is necessary in order to make sure that each process completes a reasonable number

[4] Given a chance of p percent that an operation is a read, it is not true that at any time p percent of the processes are readers: p must be weighted by the time a reader takes to acquire and release a lock relative to the time a writer takes.

of writes. When the first process finishes, we stop measuring throughput so that the drop in contention (and resulting increase in throughput) as processes finish is factored out of our results.

The last parameter that we varied is the time spent holding the lock. In one set of experiments, each process spends 0 cycles holding the lock. This measures how the algorithms scale at the highest contention levels, and also demonstrates how the overhead for lock acquisition and release scales. In the other set of experiments, each process spends 50 cycles holding the lock; this is intended to model a simple system where each process spends a relatively small amount of time holding a lock. We did not vary the time spent outside of the lock, since the effect of that would be equivalent to having fewer processes accessing the lock.

3.1 Existing Algorithms

We compared our two new algorithms with two "standard" algorithms: a *monitor-based* algorithm and an *MCS* algorithm. The first algorithm is a simple extension to a monitor-based reader-writer lock [3]; it is intended to model a very simple implementation of a reader-writer lock. The *MCS* algorithm is the fair version of the MCS reader-writer lock [15].

3.2 Monitor-based algorithm

Figure 3 outlines the structure of a reader-writer lock based on a monitor. A reader that requests a *monitor-based* lock must first acquire the monitor, which is implemented as a semaphore. It then checks the writer bit; if it is 0, the reader increments the reader count and releases the monitor; otherwise, it enqueues itself on the reader queue, releases the monitor, and waits. In order to release a *monitor-based* lock, a reader must acquire the monitor, decrement the reader count, and release the monitor. If the process finds that it is the only reader (i.e., the number of readers is 0 after the decrement), it wakes up a writer and sets the writer bit before releasing the monitor.

A writer that requests a *monitor-based* lock must first acquire the monitor. If there is already a writer, or if there are any readers, the writer enqueues itself on the writer queue. Otherwise, it sets the writer bit and releases the monitor. In order to release a *monitor-based* lock, a writer must acquire the monitor, clear the writer bit, and then release the monitor. If there are waiting readers, the writer wakes them all, and sets the reader count appropriately; if there are no waiting readers and there is a waiting writer, it wakes that writer, and sets the writer bit.

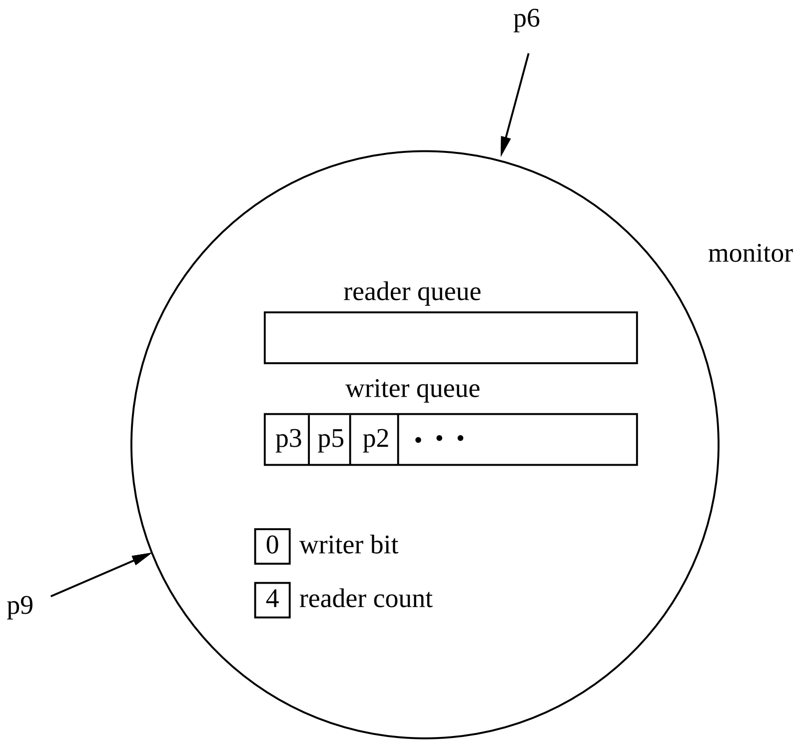

Figure 3: Monitor lock

3.3 Fair Queue

The *MCS* algorithm is one version of the MCS reader-writer lock [15]; it can be viewed as an extension of a monitor lock where the "monitor" is controlled by using the *compare-and-swap* primitive. The *MCS* algorithm is fair because it is FIFO; processes are allowed to run in the order in which they are enqueued. Although Mellor-Crummey and Scott avoid creating significant network traffic with their algorithm, they impose two requirements on processes: there is still serialization to acquire a lock, and *every* process must go over the network to acquire a lock.[5] Our algorithms attempt to avoid imposing these two requirements, and in doing so achieve better performance.

4 Performance Evaluation

Figures 4–15, which were produced by the Proteus simulator, illustrate the results of our experiments. Throughput is measured in iterations completed per 1000 machine cycles: an iteration consists of a lock

[5] These requirements are present because the *MCS* lock is an extension of the MCS mutual exclusion lock, and so retains most of its characteristics.

acquisition followed by a release. Even though we performed our experiments on a completely different platform than Mellor-Crummey and Scott's, it is important to note that the behavior of their algorithm that we observed is very similar to what is presented in their paper [15].

4.1 Comparing Lock Overhead

Figures 4–9 illustrate how the various algorithms scale when processes spend 0 cycles inside the lock; this demonstrates how the overhead of lock acquisition and release affects throughput. Figure 4 demonstrates that both of our algorithms scale linearly with 100% readers, whereas the *MCS* and the *monitor-based* locks do not allow any real parallelism among readers in their access to the lock; we cut the graph off so that the data for the *monitor-based* and *MCS* algorithms could be seen.

Figure 5 demonstrates that the *static* algorithm does not scale well, even with a small percentage of writes, because a writer incurs overhead proportional to the number of processors. With a small number of processors, a writer does not have to acquire many semaphores; however, as the number of processors increases, the throughput of acquiring a write-lock decreases dramatically, and becomes a limiting factor. The *dynamic* algorithm continues to perform quite well: with 64 or more processors, it achieves a throughput that is an order of magnitude higher than that achieved by the *MCS* algorithm.

As Figure 6 illustrates, the *static* algorithm does not scale when we decrease the ratio of reads to writes. The throughput of writers begins to dominate the curve; the shape of the curve begins to look like $1/n$, where n is the number of processors. This makes sense, as the latency for a writer increases linearly with n. The throughput curve for our *dynamic* algorithm also begins to flatten; we do not get an increase in throughput as the number of processors increases. As the number of processors increases, it is more likely that any process is performing or waiting to perform a write, since the writer latency increases; the throughput of writes becomes the limiting factor for total throughput.

We can see from Figure 7 that the *MCS* algorithm begins to outperform the *dynamic* algorithm as the number of writes becomes substantial. However, the *dynamic* lock is within a constant factor of the *MCS* lock, and these factors are larger than they should be, as we did not work hard to tune our implementation of the *dynamic* lock. For example, when a process reads the valid list, it performs a remote read for every element in the list. A more efficient implementation would read the list in larger blocks, which would result in lower latency and less network contention. For this reason, we believe that even at a lower percentage of reads, the performance of the *dynamic* lock could be comparable to that of the *MCS* lock.

Figures 8 and 9 demonstrate that as the number of writers increases, the *MCS* algorithm performs even better relative to the others. This is to be expected, since our algorithms are designed to take advantage of a large number of readers. Interestingly, the *dynamic* algorithm performs at least as well

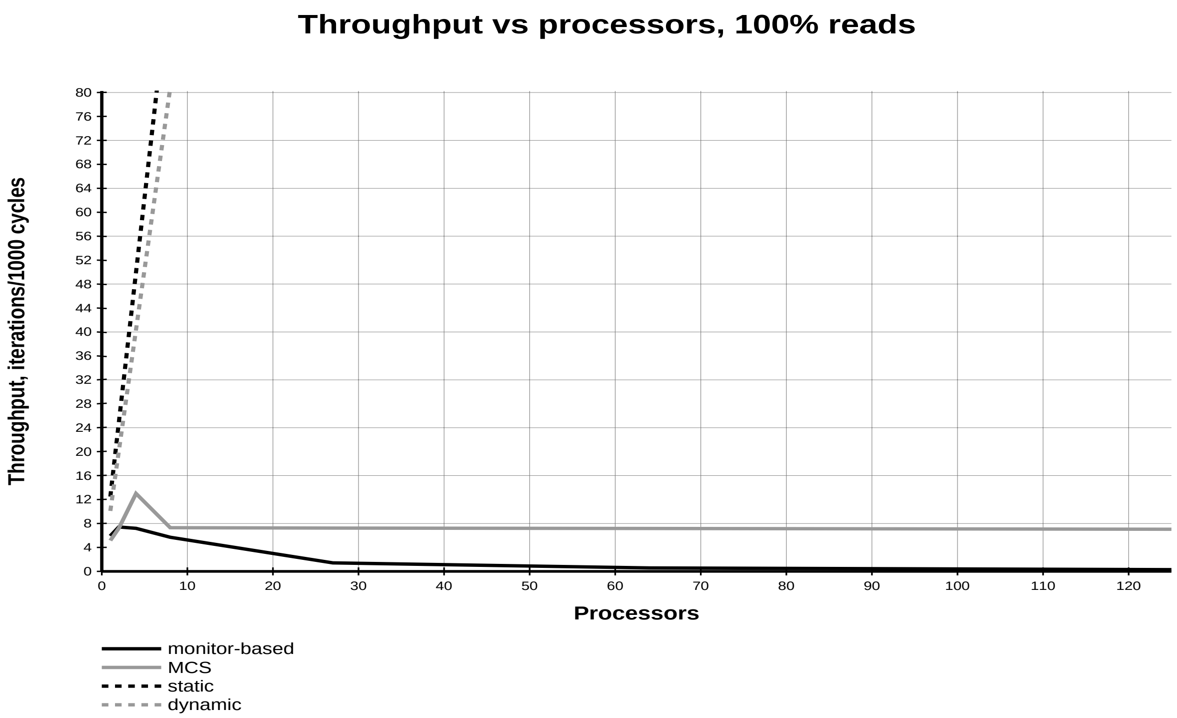

Figure 4: Scalability of Reader-Writer Lock Algorithms at 100% reads, 0% writes, 0 cycles in lock

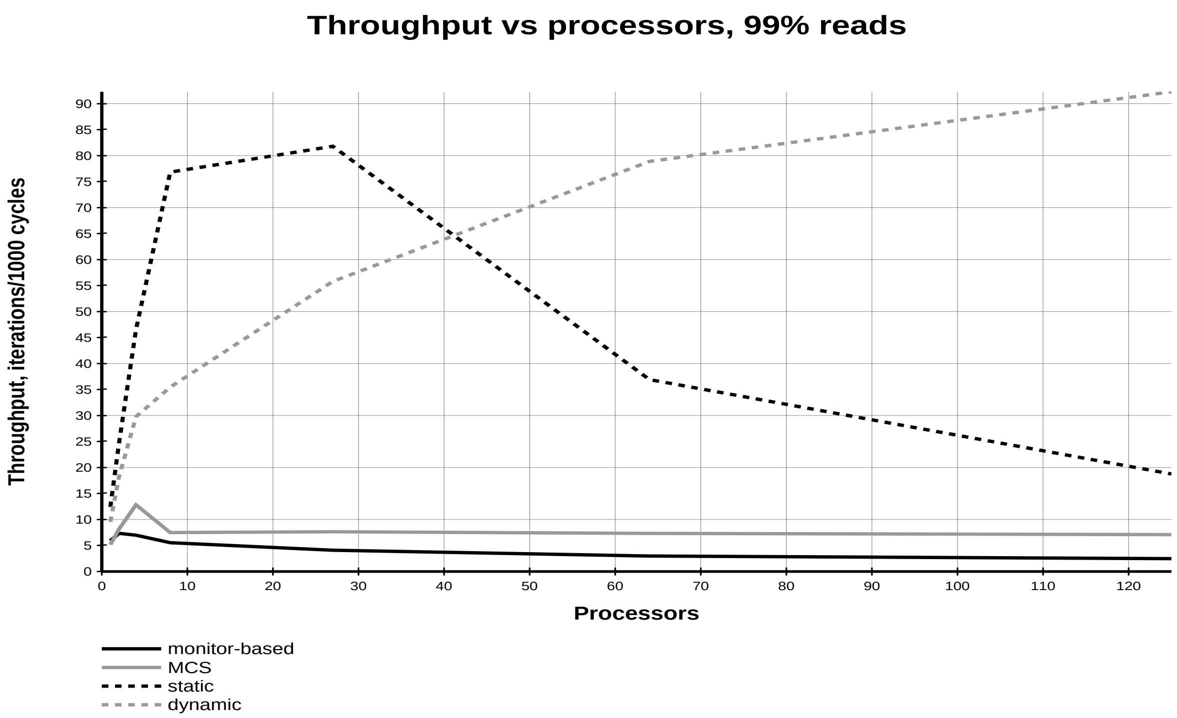

Figure 5: Scalability of Reader-Writer Lock Algorithms at 99% reads, 1% writes, 0 cycles in lock

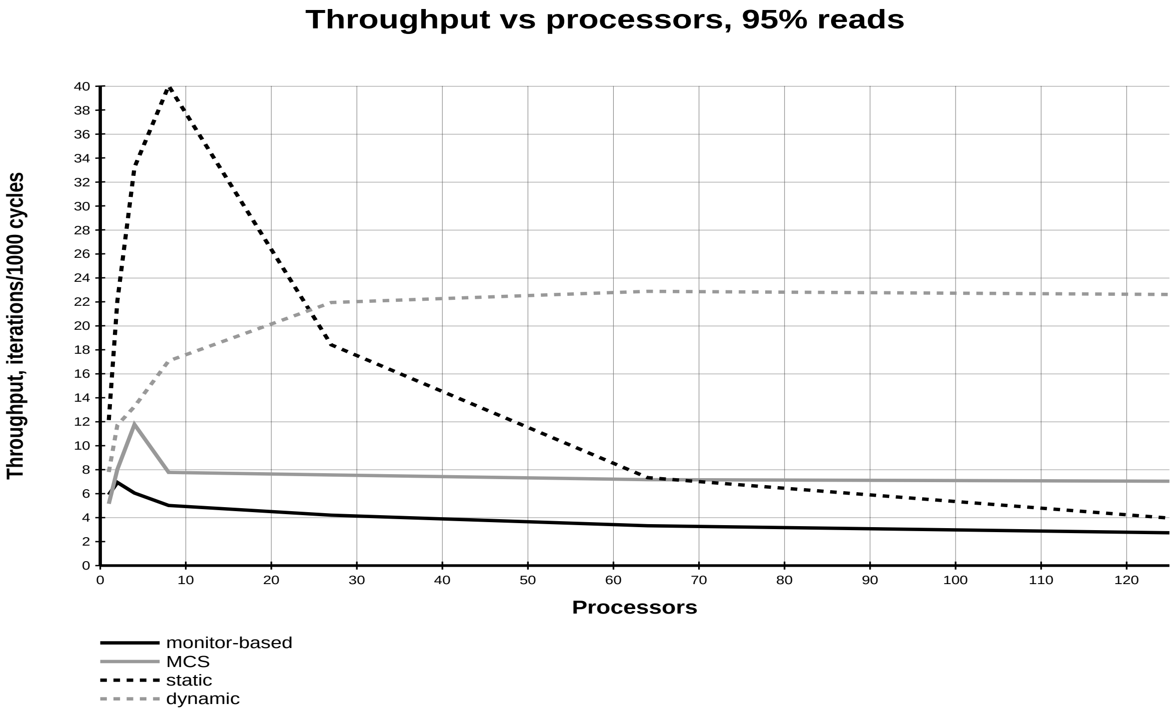

Figure 6: Scalability of Reader-Writer Lock Algorithms at 95% reads, 5% writes, 0 cycles in lock

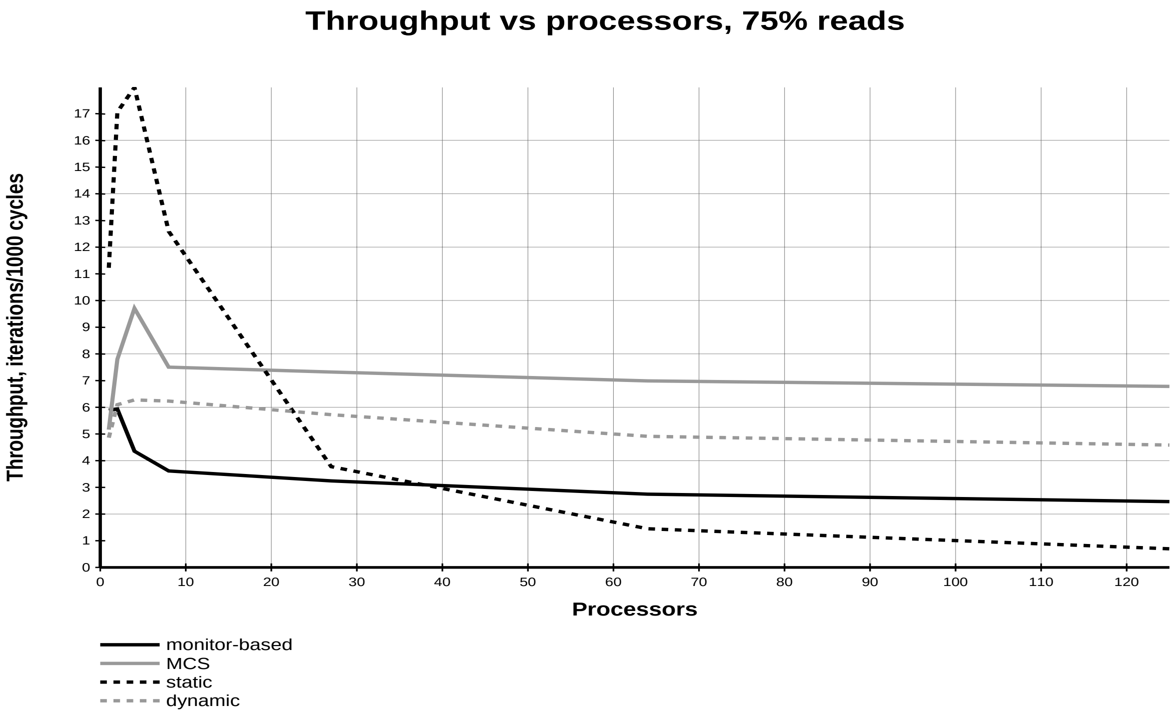

Figure 7: Scalability of Reader-Writer Lock Algorithms at 75% reads, 25% writes, 0 cycles in lock

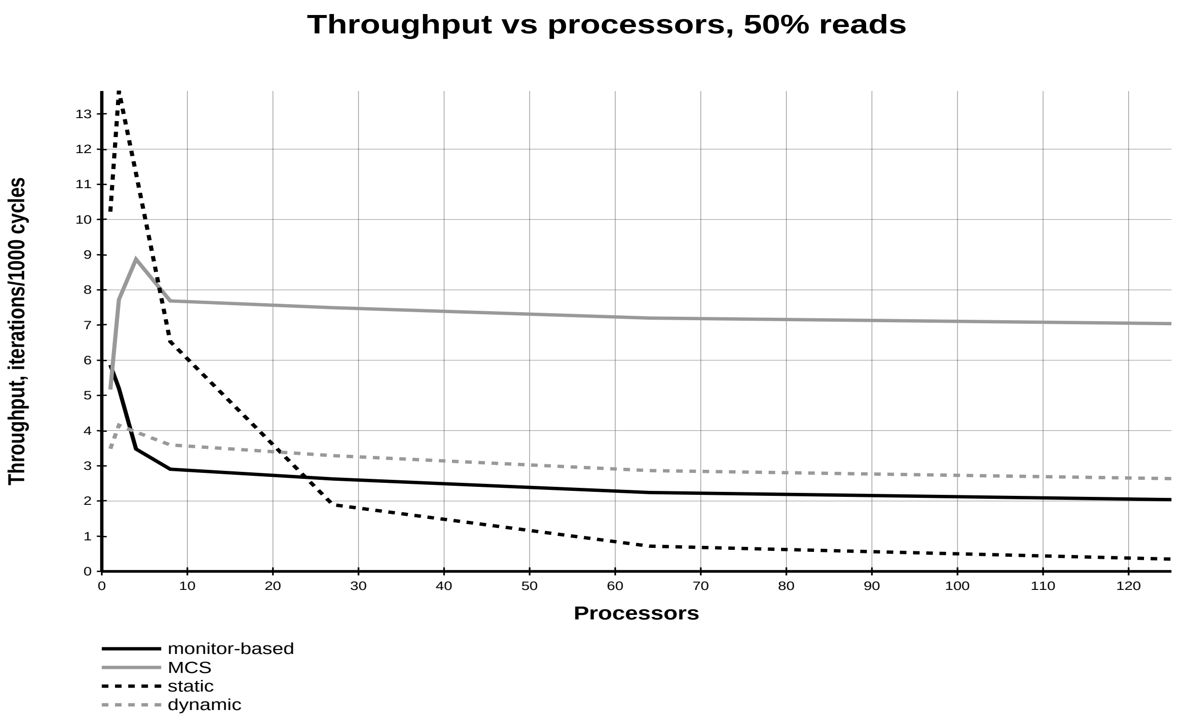

Figure 8: Scalability of Reader-Writer Lock Algorithms at 50% reads, 50% writes, 0 cycles in lock

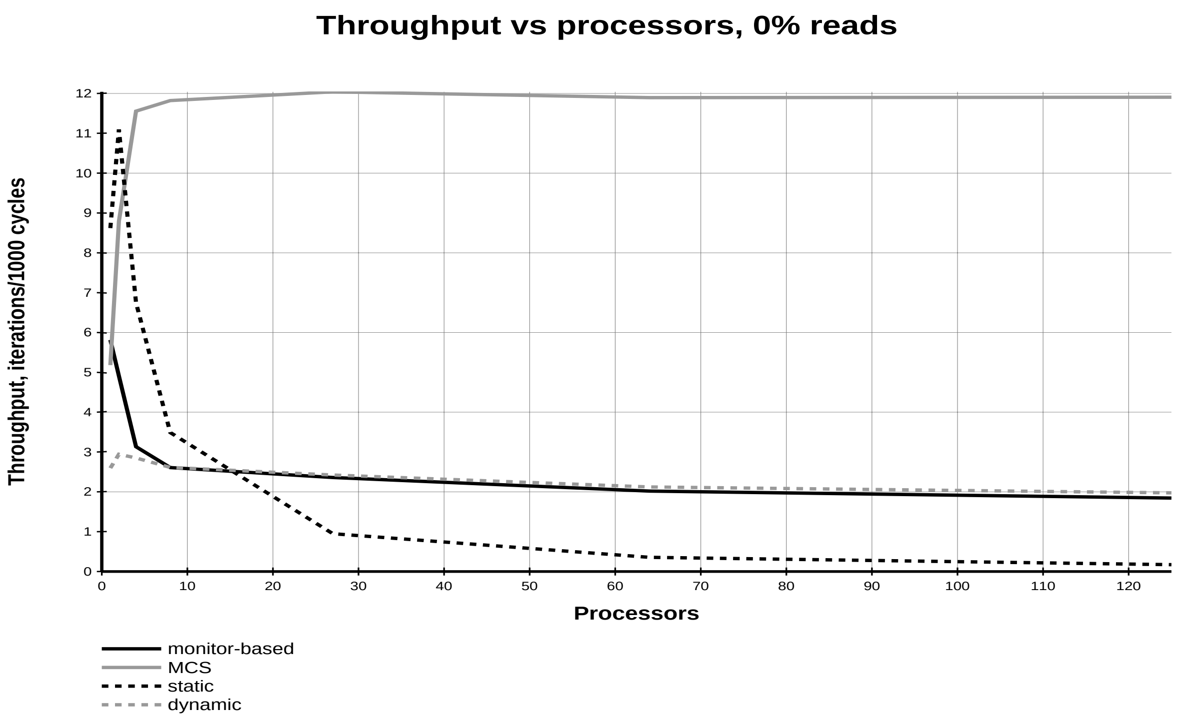

Figure 9: Scalability of Reader-Writer Lock Algorithms at 0% reads, 100% writes, 0 cycles in lock

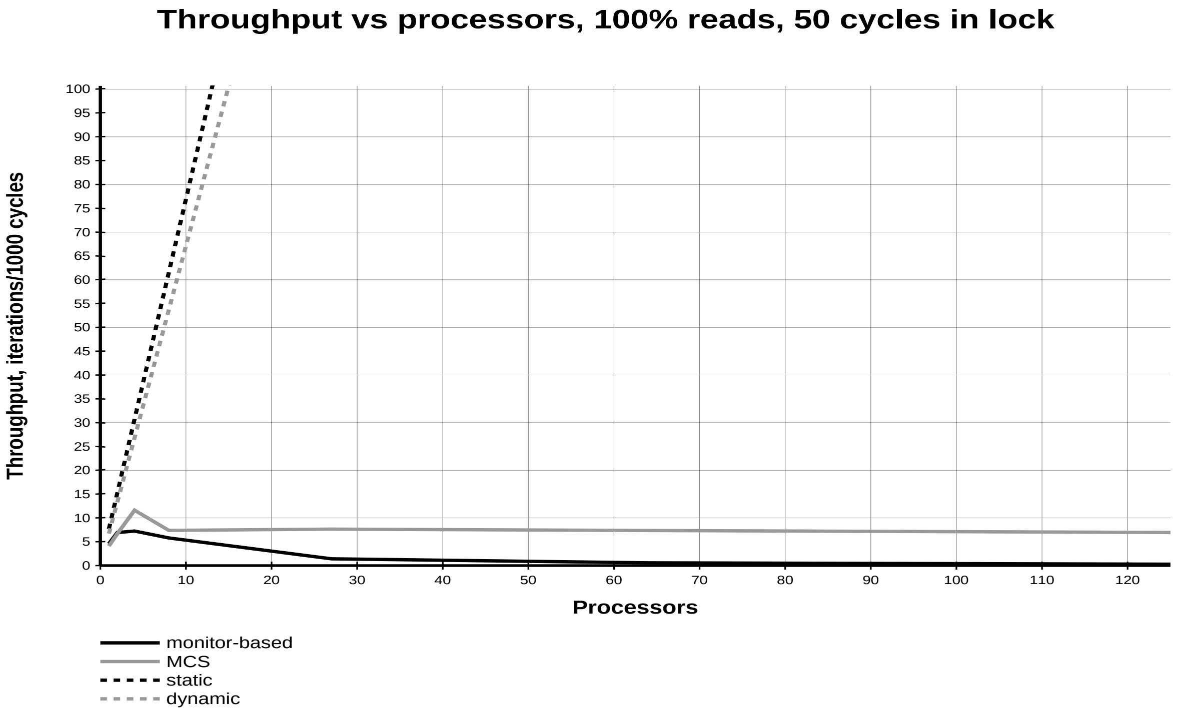

Figure 10: Scalability of Reader-Writer Lock Algorithms at 100% reads, 0% writes, 50 cycles in lock

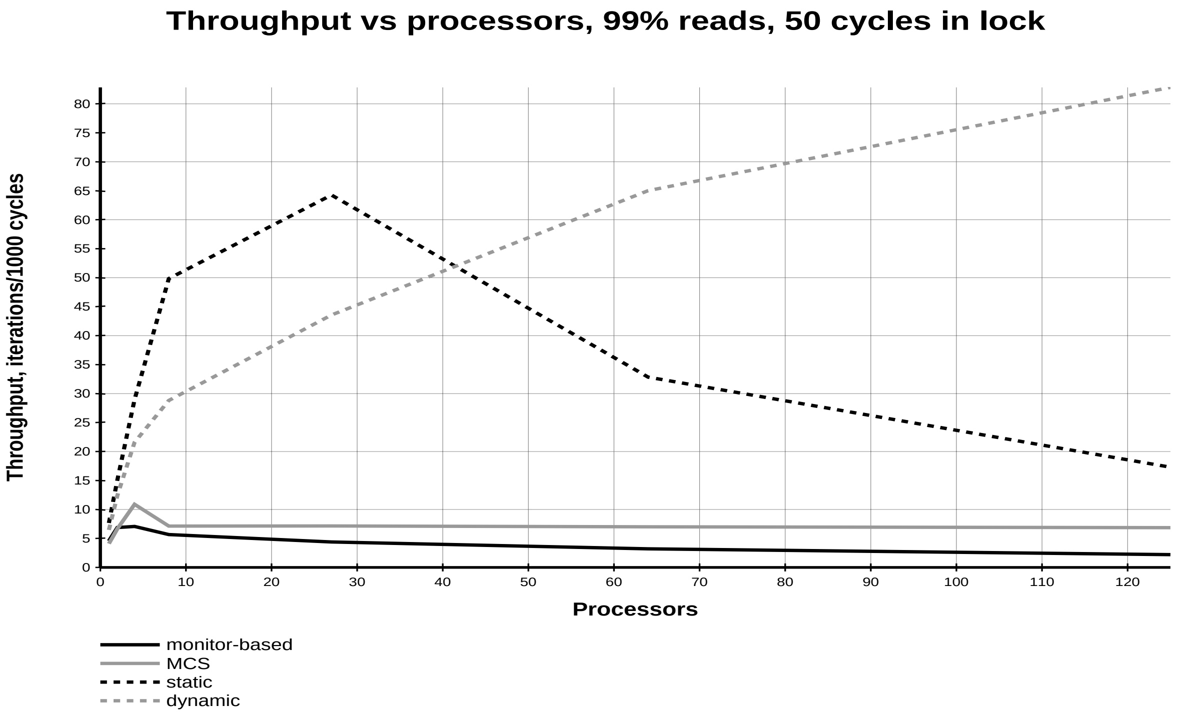

Figure 11: Scalability of Reader-Writer Lock Algorithms at 99% reads, 1% writes, 50 cycles in lock

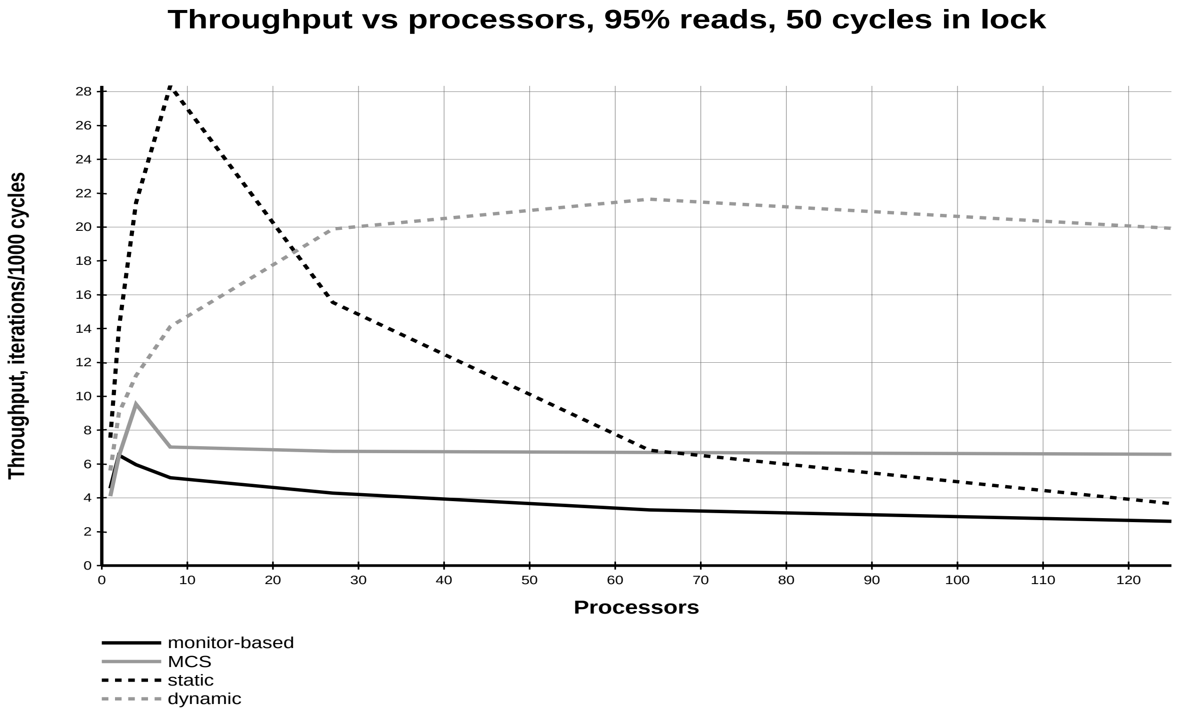

Figure 12: Scalability of Reader-Writer Lock Algorithms at 95% reads, 5% writes, 50 cycles in lock

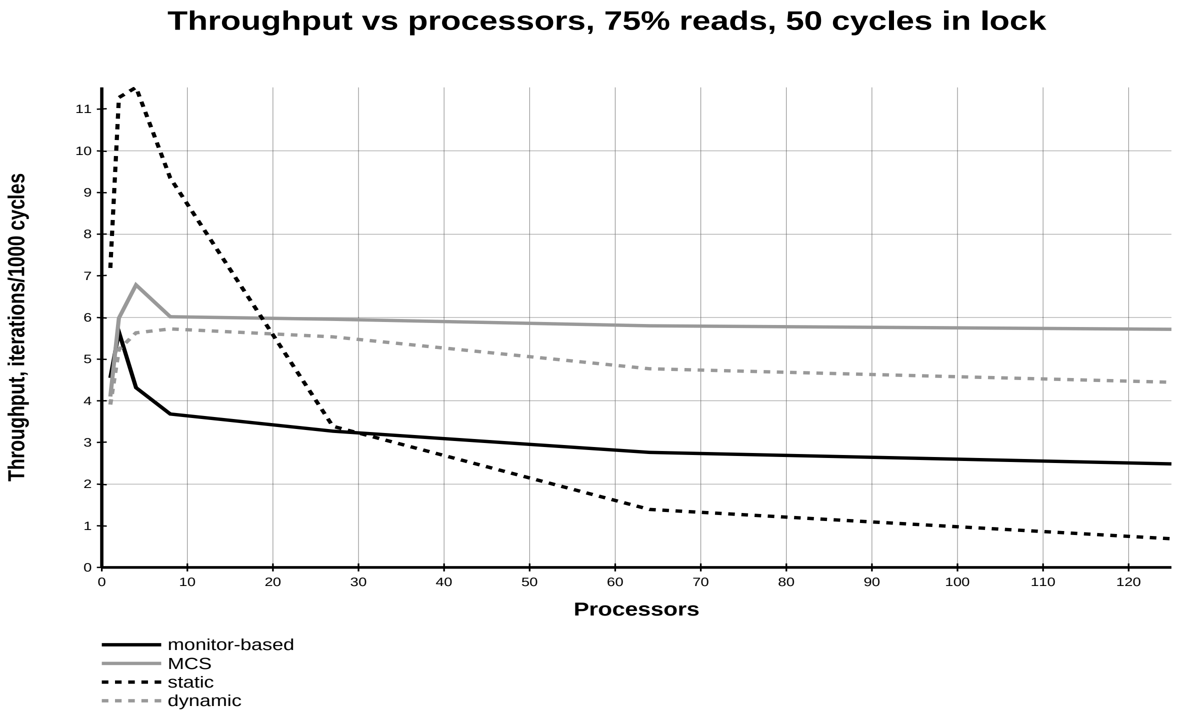

Figure 13: Scalability of Reader-Writer Lock Algorithms at 75% reads, 25% writes, 50 cycles in lock

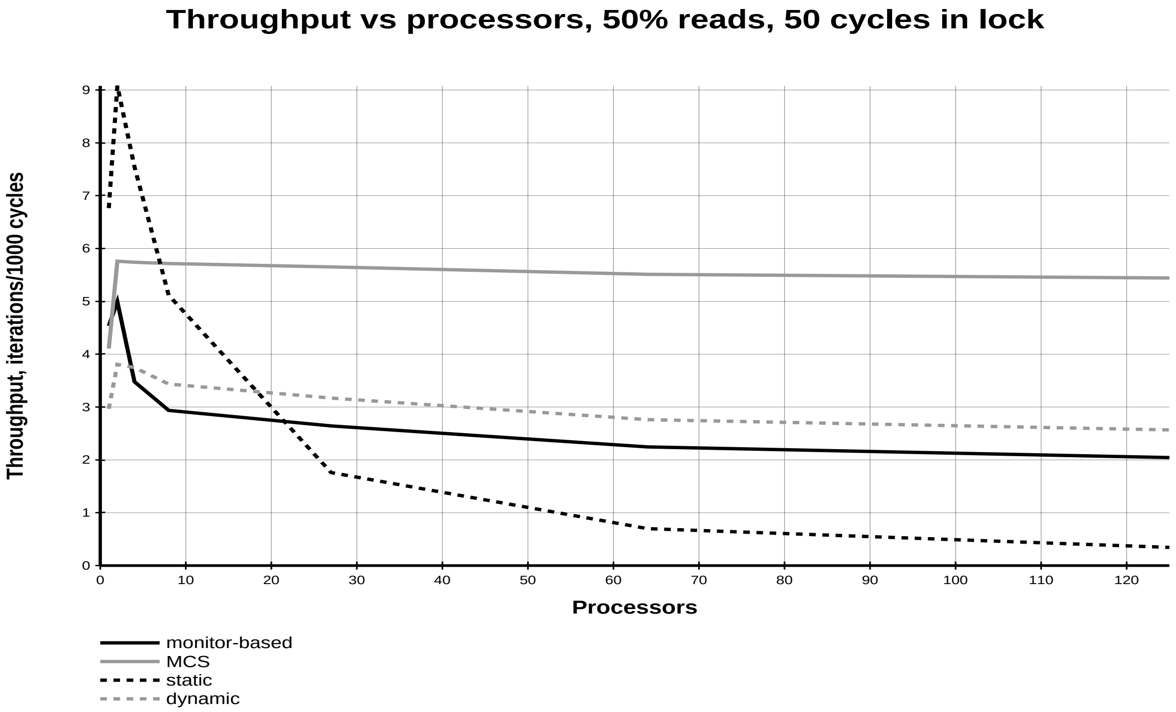

Figure 14: Scalability of Reader-Writer Lock Algorithms at 50% reads, 50% writes, 50 cycles in lock

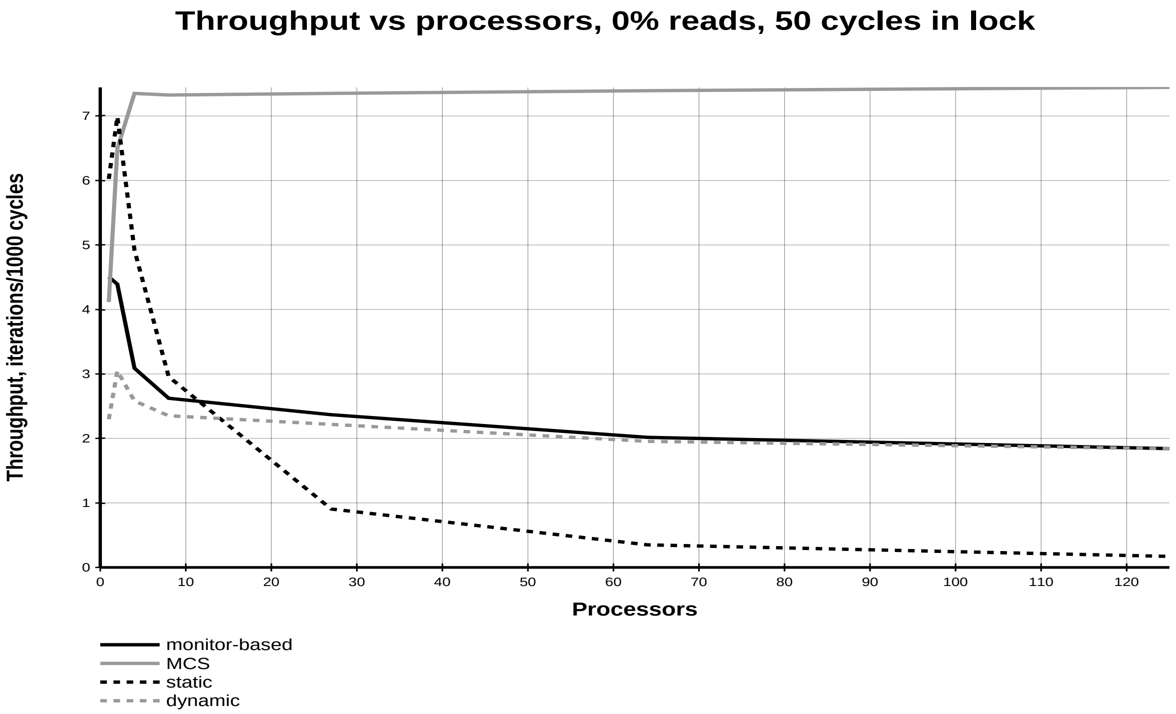

Figure 15: Scalability of Reader-Writer Lock Algorithms at 0% reads, 100% writes, 50 cycles in lock

as the *monitor-based* algorithm, even with 0% readers.

4.2 Performance Under A Synthetic Load

Figures 10–15 illustrate how the various algorithms scale when processes spend 50 cycles inside the lock. This models a more realistic situation, where processes do spend some time holding a lock. The shapes of the throughput curves in these figures are very similar to those in Figures 4–9, and we can draw similar conclusions.

5 Related Work

This work is very similar to work in distributed filesystems [6, 17], although the thrust is different. Our goal is to increase throughput in multiprocessors, whereas the main motivation in distributed systems has typically been to avoid the latency of remote lock requests. Burrows' MFS filesystem [6] uses a locking scheme that resembles our *dynamic* lock, where a central server keeps track of all of the processes with read locks. The Vaxcluster system [17] provides a distributed lock manager that allows a more general locking semantics than reader-writer synchronization; its structure is also similar, in that each resource has an associated lock-manager that keeps track of the outstanding locks for that resource.

Distributed systems also deal with dropped messages, network partitions, and processor failures; however, most multiprocessor systems are not designed to deal with such occurrences. The use of time-based expiration of locks has been explored by Gray and Cheriton as a means of dealing with fault-tolerance [10]: a process that requests a lock gets a *lease*, which expires after a certain time. It would be interesting to combine such a mechanism with the *dynamic* algorithm, as we expect that fault-tolerance will become more important as multiprocessors scale.

There is also some similarity between this work and research on cache coherence. The *dynamic* lock can be viewed as directly implementing a directory-based cache consistency scheme for a reader-writer lock [1]: the valid list is analogous to a directory of possible readers. Similarly, the *static* lock can be viewed as generalizing a bus-based (broadcast) cache consistency scheme.

Recent work has explored two-phase (spin, then block) algorithms for waiting in multiprocessors [11, 13]. This work has shown that two-phase algorithms perform better than pure spinning or blocking, and that some two-phase strategies perform within a constant factor of the offline optimal decision to spin or block. In generalizing our algorithms to handle multiple processes per processor, it would be very useful to incorporate this work.

6 Conclusion

We have experimented with two new algorithms for reader-writer synchronization, in the hope that the opportunity for parallelism between readers would allow us to achieve higher throughput than existing algorithms. Although the *static* algorithm does not exhibit better performance (except when there are very few writes), the *dynamic* algorithm achieves substantially higher throughput and better scalability than current algorithms.

In environments where reads are a high percentage of lock requests, the *dynamic* algorithm performs significantly better than the *MCS* lock. When writes are a significant proportion of lock requests, the *MCS* lock performs slightly better. This is to be expected, for as the percentage of writes increases, a reader-writer lock increasingly behaves like a mutex. In highly parallel systems, it is likely that writes to heavily shared memory will be infrequent; if there are many such writes in an application, the parallelism available is likely to be very low.

It would be useful to find analytic bounds on the performance of reader-writer locks; we could then evaluate in a more absolute sense how the *dynamic* lock performs. We actually designed another algorithm that uses combining-tree techniques: we intended this algorithm to be more adaptive than the *dynamic* algorithm. Unfortunately, this tree-based algorithm always performs worse than the dynamic algorithm. We are investigating the possibility of improving this algorithm; in particular, the use of blocking instead of spinning may help.

An important implication of our research is that the design of an algorithm should take into account the semantics of the problem. The *MCS* lock does not take advantage of the differences in semantics between reader-writer synchronization and mutual exclusion; as a result, the throughput that it achieves for readers does not scale. Our results suggest that significant performance advantages can be obtained by paying careful attention to the semantics of synchronization primitives.

7 Acknowledgments

We would like to thank Eric Brewer and Chris Dellarocas for their help with Proteus, as well as for providing Proteus as a platform. Thanks to Eric Brewer, Adrian Colbrook, Beng-Hong Lim, Sharon Perl, and Carl Waldspurger for their help in greatly improving this paper, and to Bob Gruber and Paul Wang for their time in discussing this research.

References

[1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on*

Computer Architecture, May 30–June 2, 1988, pages 280–289.

[2] Thomas E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. In *IEEE Transactions on Parallel and Distributed Systems*, 1(1), January 1990, pages 6–16.

[3] Toby Bloom. Synchronization Mechanisms for Modular Programming Languages. MS thesis, MIT, MIT/LCS/TR-211, January 1979.

[4] Eric A. Brewer. Aspects of a High-Performance Parallel-Architecture Simulator. MS thesis, MIT, December 1991.

[5] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. Proteus: A High-Performance Parallel-Architecture Simulator. MIT, MIT/LCS/TR-516, September 1991.

[6] Michael Burrows. Efficient Data Sharing. PhD dissertation, University of Cambridge, September 1988.

[7] Adrian Colbrook, Eric A. Brewer, Chrysanthos N. Dellarocas, and William E. Weihl. An Algorithm for Concurrent Search Trees. In *Proceedings of the 1991 International Conference on Parallel Processing*, August 12–16, 1991, Volume III, pages 138–141.

[8] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent Control with "Readers" and "Writers". *Communications of the ACM*, 14(10):667–668, October 1971.

[9] Chrysanthos N. Dellarocas. A High-Performance Retargetable Simulator for Parallel Architectures. MS thesis, MIT, MIT/LCS/TR-505, June 1991.

[10] Cary G. Gray and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 3–6, 1989, pages 202–210.

[11] Anna R. Karlin, Kai Li, Mark S. Manassee, and Susan Owicki. Empirical Studies of Competitive Spinning for Shared-Memory Multiprocessors. To appear in *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 13–16, 1991.

[12] Leslie Lamport. A Fast Mutual Exclusion Algorithm. Research Report 7, Digital Systems Research Center, October 31, 1986.

[13] Beng-Hong Lim and Anant Agarwal. Waiting Algorithms for Synchronization in Large-Scale Multiprocessors. MIT VLSI Memo 91-632, July 1991.

[14] John M. Mellor-Crummey and Michael L. Scott. Synchronization Without Contention. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 8–11, 1991, pages 269–278.

[15] John M. Mellor-Crummey and Michael L. Scott. Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, April 21–23, 1991, pages 106–113.

[16] M. Raynal. *Algorithms for Mutual Exclusion*. MIT Press, 1986, pages 90–97.

[17] William E. Snaman, Jr., and David W. Thiel. The VAX/VMS Distributed Lock Manager. In *Digital Technical Journal*, Digital Equipment Corporation, Number 5, September 1987, pages 29–44.