A Low-Latency Scalable Locking Algorithm for Shared Memory Multiprocessors

Amitabh Dave, Nayeem Islam and Roy H. Campbell
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract

In this paper we deal exclusively with spin locks for small- to medium-scale shared memory multiprocessors. The performance of a lock depends on the contention for the critical section being protected by the lock. Static selection of the most appropriate lock for a critical section is difficult because the level of contention can vary dynamically. We verify that no current lock has the best performance for all critical sections. We present a new lock, called the *shared array* lock, which alleviates this problem. The *shared array* lock belongs to the class of locks based on queuing of processors waiting for the lock. Unlike existing queue-based locks which suffer from high latency in low contention situations, experimental results show that the *shared array* lock has low latency and good scalability.

1 Introduction

Spin locks are widely used in operating systems and multi-threaded applications to protect critical sections, permitting mutually exclusive access to shared data structures. Locks are used in the scheduling, memory management and process management subsystems of most modern operating systems[11]. Many parallel applications including *Water*, *Barnes Hut*, *Locus-Route* and *Cholesky* use spin-locks for exclusive access to shared variables and task queues[12]. The operations to acquire and release locks may be invoked a large number of times. The overhead of performing these operations can have a significant effect on application performance. It is essential that operating systems and thread libraries provide efficient locking techniques for critical sections in operating systems and applications.

Critical sections are characterized by their size and expected contention. The contention for a critical section is a measure of the number of processes trying to enter the section. While most operating system critical sections have low contention[13], some may experience high or even variable contention. For example, an operating system may use gang scheduling for multi-threaded applications like the Fast Fourier Transform[7]. The gang scheduler first waits for the required number of processors to become free. Each processor then selects an application thread from the scheduler queue. Contention for the scheduler queue is likely to be high during this period. Similarly, critical sections in applications also have differing levels of contention: Locks in *Water* are accessed by half the processors used by the application, *Locus-Route* and *Cholesky* experience low contention and the level of contention for *Barnes Hut* varies with the number of processors[12].

The size of a critical section is a static parameter that quantifies the time spent executing the section. For very small critical sections it is inefficient to use locks with large locking overhead unless contention is high and some of the locking overhead for an arriving processor can be overlapped with the execution of the processor holding the lock. Larger critical sections make this overlap more feasible even in low contention situations. The performance of some locking techniques varies with critical section size.

In this paper we investigate spin locks for small- to medium-scale (2–16 processors) shared-memory multiprocessors. We verify that no current lock has the best performance for all critical sections and that an operating system or thread library must therefore offer a selection of locks with different performance characteristics. This allows application and operating system developers to choose the correct lock based on expected contention and critical section size. This approach works well except for critical sections with dynamically varying or data dependent contention which forces users to compromise and choose a lock which

works well in the average case. We present a new lock as a solution to this problem. Experimental results show that this new lock, the *shared array* lock, has low latency and scales well. It is therefore a good choice for critical sections with variable contention. The lock also compares favorably with existing locks in terms of scalability, making it useful for critical sections with high contention.

The rest of this paper is organized as follows. Section 2 discusses previous work on spin locks for shared-memory multiprocessors. Section 3 presents performance data on several well known locking algorithms. We describe the design of the *shared array* lock in section 4. We also document its performance and include a proof that the algorithm is correct. Section 5 states our conclusions.

2 Related Work

Efficient algorithms for implementing spin locks on shared memory multiprocessors minimize bus or interconnection network contention to reduce overhead[1, 4, 9]. Scott and Mellor-Crummey propose a busy-wait synchronization algorithm using list-based queues that requires one global memory access per lock acquisition[9]. Their algorithm uses the compare-and-swap primitive. Excess bus or network traffic is avoided by spinning on a cached copy of the lock variable. This scheme works best for cache-coherent machines and NUMA machines. The authors also evaluate the performance of several alternative schemes including test-and-set, test-test-and-set and algorithms using backoff. They conclude with a set of recommendations for choosing the appropriate lock for a critical section.

Anderson's algorithm[1] and the Sequent locks[4] use an array implementation of queues. Anderson also proposes a backoff algorithm, in which retries to acquire a lock are separated by statically or dynamically determined intervals. These algorithms are evaluated using simple benchmarks to measure the lock latency and contention in the communication medium. The author experimentally proves that the array lock has the best scalability among all the locks he investigated.

A study of the effects of operating system synchronization on applications concludes that contention for scheduler locks is minimal for small-scale (up to 4 processors) shared memory multiprocessors[13]. Contention for a larger number of processors or in the presence of gang scheduling is not studied, though the authors foresee an increase in contention as the number of processors increases.

The Synthesis kernel is an operating system which does not use locks at all for its critical sections. Massalin and Pu[8] show that some operating system synchronization can be handled by limiting shared data to two words and using the atomic double compare-and-swap instruction. Longer critical sections are handled using lock-free synchronization, a variant of wait-free synchronization[5]. It is not clear what impact the overhead of the lock-free technique has on application performance compared to spin locking overhead. In a more recent paper[6], Herlihy shows that the wait-free technique performs worse than spin locks with backoff by a factor of two, which illustrates the performance tradeoff required for fault tolerance.

Mukherjee and Schwan [10] present a study of reconfigurable locks and consider strategies for dynamically configuring locks as blocking or busy-waiting.

Our work differs in that we present the design of a new queue-based lock, the *shared array* lock. This lock has lower latency than other queue-based schemes and also scales well as contention increases. Unlike some other locks, the *shared array* lock can be efficiently implemented using the test-and-set instruction, a common atomic instruction available on commercial processors. Our design also tolerates preemption of spinning processors, which is unacceptable in most other queue-based schemes.

3 Performance of Locking Algorithms

In this section we analyze the performance of several well known locking schemes. We used the Encore Multimax 320 as the testbed for our locking experiments. The Multimax has 16 NS32332 processors, each with a private 64KB cache and access to 32MB of shared main memory[2]. Hardware support for synchronization is limited to the test-and-set primitive[3]. The Multimax uses a *write-through with invalidate* cache protocol. The language used in our implementation is C++ with small parts of code in N32000 family assembler[3].

We consider the following locks in our experiments:

- *TASLock* This is a simple test-and-set lock in which each processor loops on the shared lock variable using the test-and-set instruction until it acquires the lock.

- *TTASLock* The test-test-and-set lock is similar to the test-and-set lock except that each spinning processor uses the test-and-set instruction only after reads of the lock variable have shown

the lock to be free. The number of atomic instructions executed for each acquire is therefore reduced.

- *TASELock* Another way of reducing the number of atomic instructions is by introducing a delay between each test-and-set, where the delay is a function of the number of failed attempts to acquire the lock. This scheme is called test-and-set with exponential backoff. The maximum delay is bounded to prevent excessive delays.

- *ArrayLock* The array lock queues processors waiting for the lock. Whenever the lock is released, the processor at the front of the queue is dequeued and allowed to enter the critical section.

Detailed descriptions of these locks can be found elsewhere[9, 1].

3.1 Performance

Two measures quantify the performance of each lock: *Latency* is the time required to acquire and release a lock. A lock is *scalable* if the overhead for lock acquire and release does not increase appreciably with increased contention for the lock. These two measures often conflict because scalability provisions increase latency.

We modified a version of the test suite of Scott and Mellor-Crummey[9] for our experiments. The test measures the time required for 100,000 lock acquire-release pairs with varying critical section sizes and numbers of processors. Figures 1 and 2 document the performance for two different critical sections: a null critical section and a section that takes 190 μs.

Our experiments show that the test-and-set lock has the lowest latency and performs the best in the presence of low contention. As contention increases, the array lock performs the best. The test-and-set lock with exponential backoff also scales well. An increase in the critical section size affects the performance of the test-test-and-set lock and test-and-set lock with exponential backoff. The test-test-and-set lock performs better for larger critical sections. This is because the tests of the locally cached copy do not generate bus traffic and the longer the critical section the more advantageous this becomes. The performance of test-and-set with exponential backoff exhibits an interesting behavior. For large critical sections, processors spinning on the lock reach the maximum exponent bound. If the number of processors is low there is a greater chance for all processors to have reached the bound and be delayed even when the lock is free. The peaks for odd numbers of processors occur because the processors which have not reached the bound try to acquire the lock more often. There is a tendency for the lock to alternate between pairs of processors leaving the last one (the odd one out) to do its share of acquires and releases serially when the others have finished. The bound used for the backoff is 16 in our implementation.

The lock behavior we document above agrees with similar experiments carried out by other researchers on different shared memory multiprocessors[9, 1]. This validates the environment we used for our tests.

4 The New *Shared Array* Lock

Figures 1 and 2 show that the most desirable performance for any lock is a combination of the performance of the test-and-set lock and the array lock. This is because the test-and-set lock has the lowest latency and the array lock has the best scalability. Our design goal for the new *shared array* lock was to achieve this performance. Since our design is based on the array lock, we first describe its design and limitations in some detail. We discuss the intuition behind our modifications to the lock and then present the detailed algorithm, an argument for correctness, and experimental results that document the lock's performance.

4.1 The Array Lock

In Anderson's algorithm[1], each arriving processor is assigned a unique sequence number using an atomic fetch-and-increment on a global variable. We refer to the sequence number of the processor allowed into the critical section as the current service number. Each processor spins waiting for its sequence number to become the current service number. When a processor leaves the critical section, it increments the current service number and hands the critical section ownership to the corresponding processor. There are at least two different mechanisms that may be used to implement the locking needed for the critical section. With write-invalidate cache coherency, the implementation may use an array of spin locks, indexed by the sequence number. With write-update cache coherency, the implementation may use a global variable to access the current service number[1].

The algorithm in Figure 3 assumes multiprocessors with write-invalidation cache coherency. To avoid cache line contention, the lock array is mapped so that

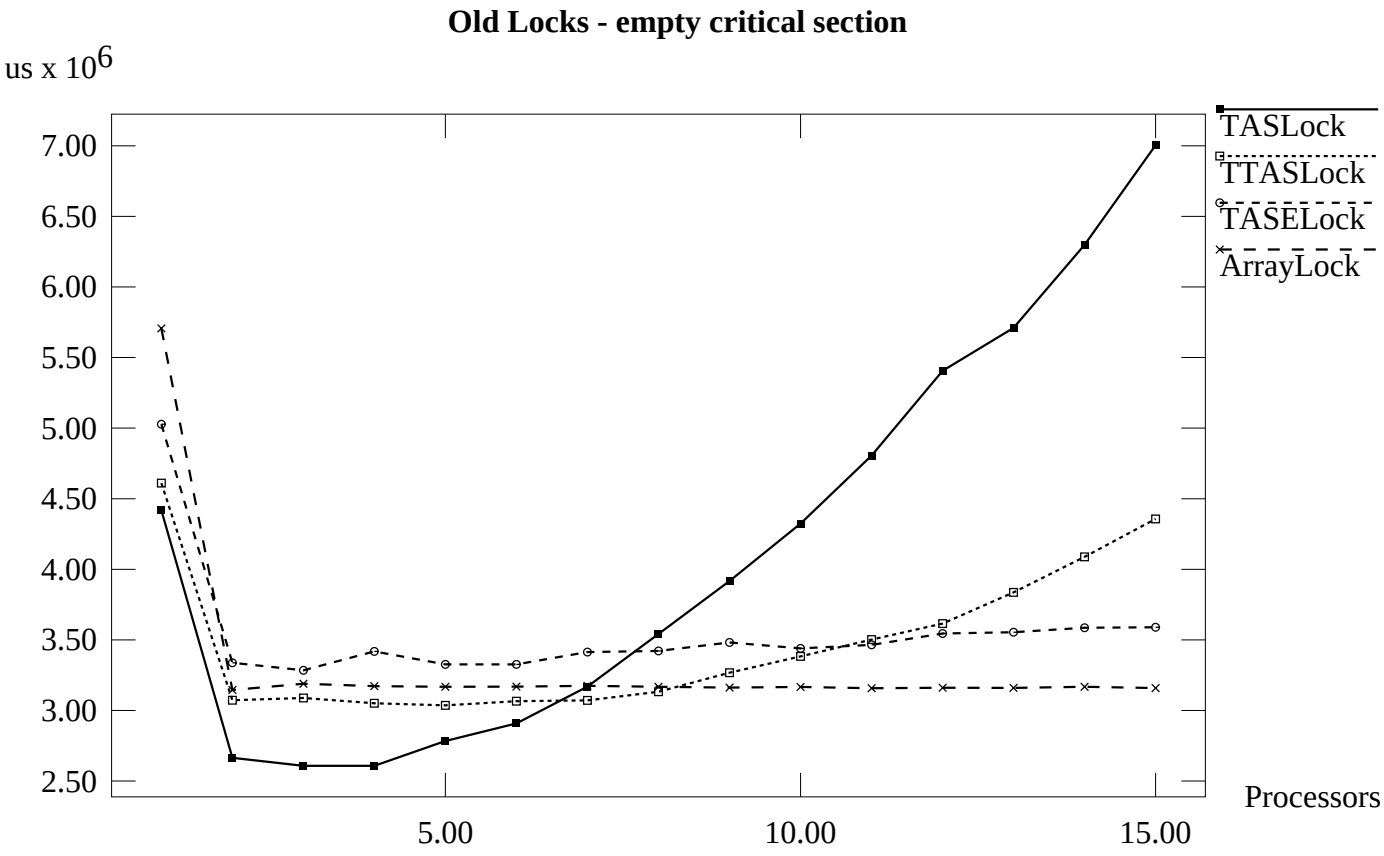

Figure 1: Time for 100000 acquire–releases

each spin lock is in a different cache line, limiting network or bus transactions to two per critical section execution (an invalidation and a read miss).

Anderson lists three limitations of his algorithm.

- *Queuing increases lock latency.* Each arriving processor must atomically fetch-and-increment a counter, check a location and re-initialize the location before entering the critical section. For critical sections with low contention, locks using a simpler technique like test-and-set have lower locking overhead.

- *Queuing exacerbates the problem of preemption of a processor while spinning.* Preemption of the processor holding the lock is undesirable for any locking technique. For methods based on queuing, preemption of *any* spinning processor is undesirable as it prevents all processors queued behind the preempted processor from entering the critical section.

- *Queuing makes it difficult for a processor to wait on multiple critical sections.* It is therefore difficult to use queuing to gain access to a resource whose control is distributed over several critical sections.

Performance numbers published by Anderson[1] support his contention that the array lock has excellent scalability. Unfortunately they also show high lock latency in low contention situations.

4.2 The New *Shared Array* Lock

The *shared array* lock reduces the latency associated with implementing the sequence number scheme. The algorithm still uses an array of locks to reduce bus traffic on machines with write-invalidation caches. The same scheme would work correctly for write-update caches, though due to the lack of an appropriate testbed we have no performance results for this case. Each arriving processor tests a global *logical lock* and if it is unlocked, the processor enters the critical section. If the logical lock is locked, then the processor retrieves a count of the number of processors

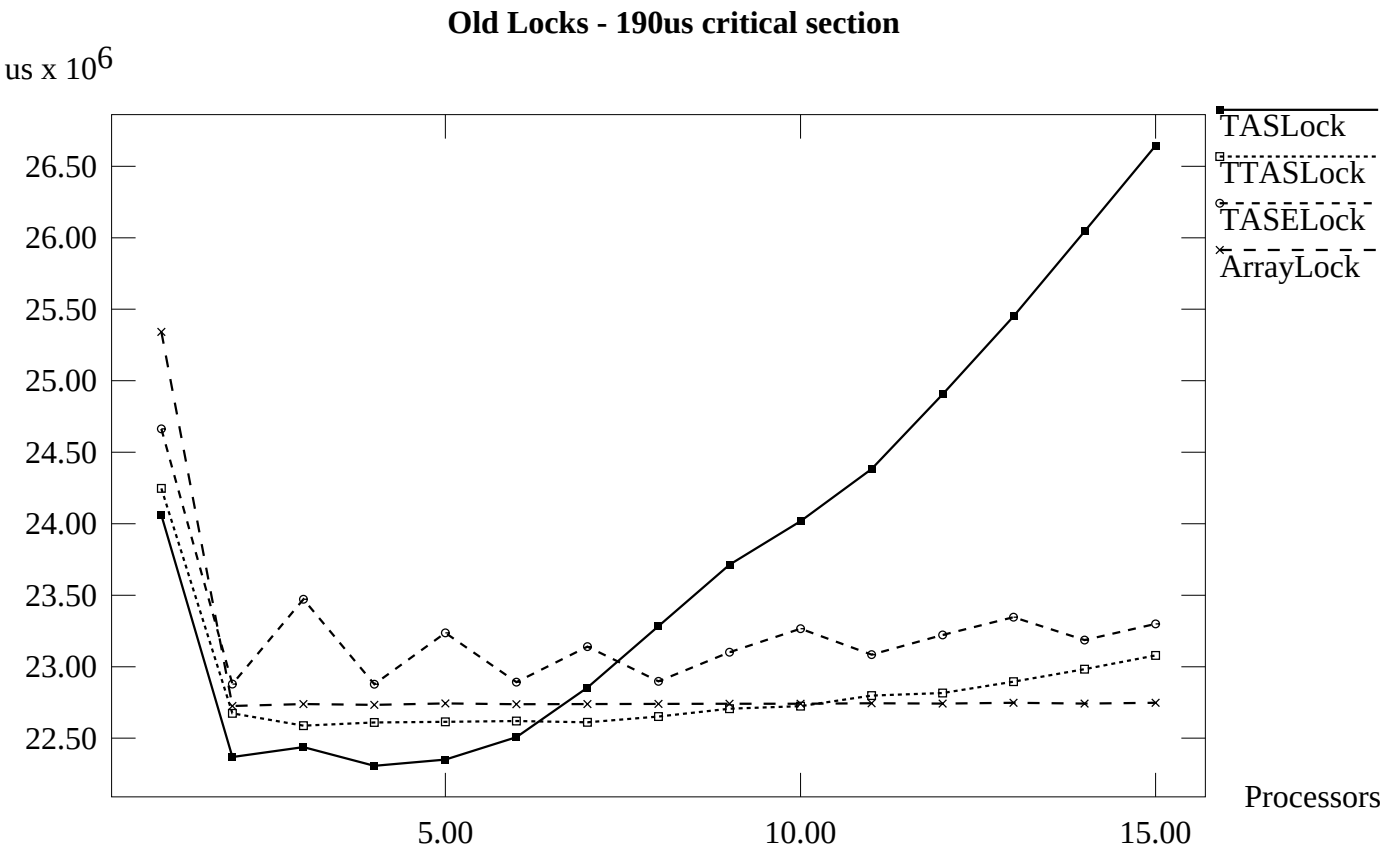

Figure 2: Time for 100000 acquire–releases

spinning on the logical lock. If the count is below a threshold value, the processor atomically increments the count and then spins on the logical lock. Otherwise, the processor searches an array of spin locks for the first lock location that is unlocked. It acquires the lock and spins waiting for another processor to release this lock.

When a processor leaves the critical section, it sets off a chain of events that advances the other processors through the shared array lock algorithm. The processor leaving frees both the logical lock and the first array spin lock. A processor spinning on the logical lock will lock it and enter the critical section. The processor spinning on the first array spin lock releases the second array spin lock and starts spinning on the logical lock. Similarly, the processor spinning on the second array spin lock releases the third array spin lock and starts spinning on the first array spin lock. In turn, each processor spinning on an array spin lock releases the next higher indexed spin lock in the array and spins on the next lower spin lock.

The new *shared array* lock is constructed to work even if processors are preempted from their current tasks while spinning on the logical or array spin locks. In such a case, since the processor is performing a different task, if its spin lock is released, it may not immediately release the next higher indexed spin lock in the array. The shared lock algorithm has each spinning processor occasionally check the spin lock it would move to when a release occurs. If the spin lock is unlocked, the processor resumes the chain of advances. This part of the algorithm may permit multiple processors to share the same spin lock in the array. However, such sharing is removed in subsequent advances of the processors.

We list the complete algorithm in Figures 5 through 7 using a C-like syntax. For convenience we use the spin lock at location 0 in the lock array as the logical lock. NUM_WAITERS is the threshold value of waiters on the logical lock and NUM_SPINS is the number of time a processor spins on its spin lock before checking the next lower indexed spin lock. The implementation of the algorithm on the Encore Multimax requires about 80 assembly language

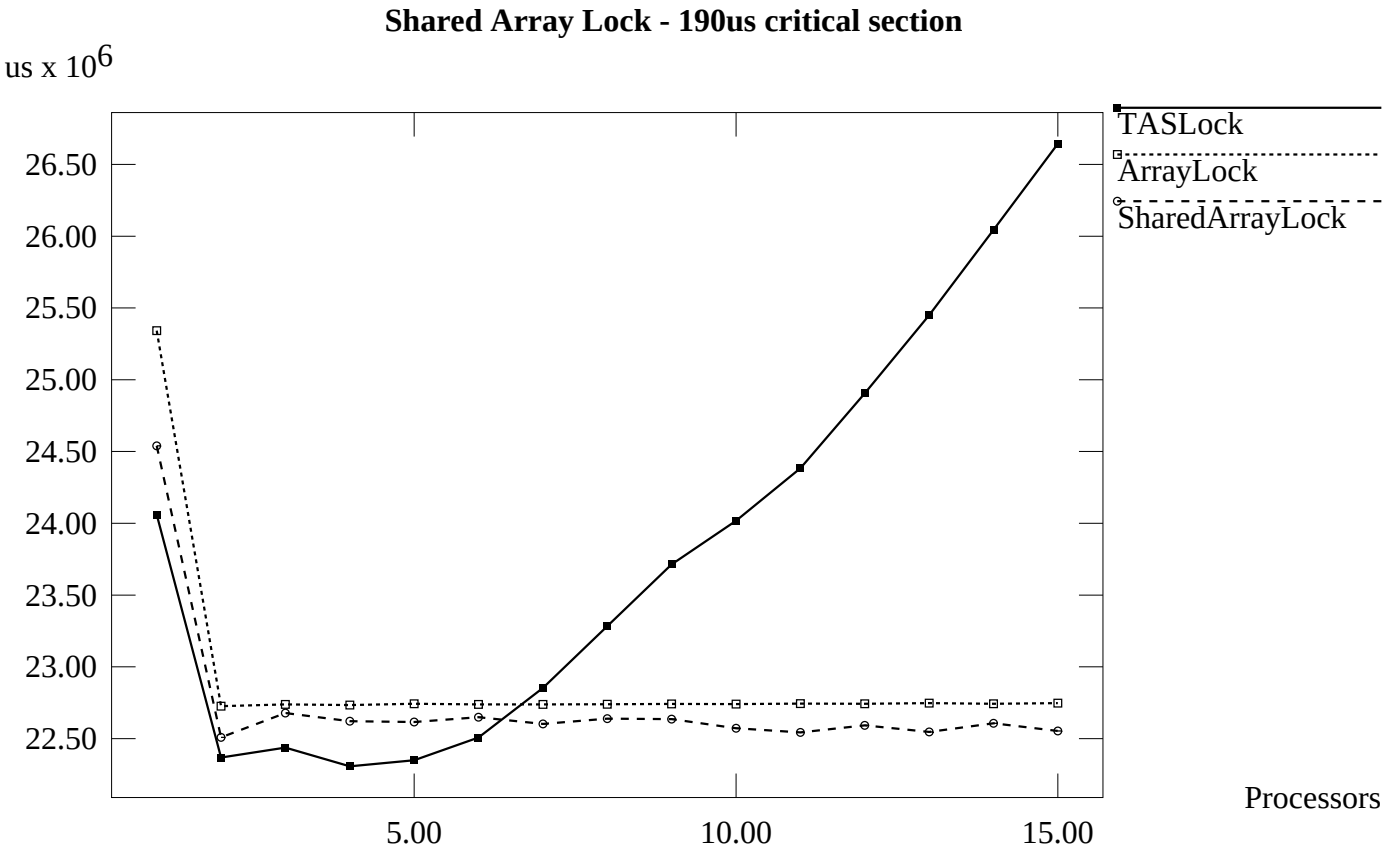

Figure 4: Time for 100000 acquire–releases

instructions.

4.3 Performance

The performance characteristics of the *shared array* lock are given in Figure 4. For these experiments the parameters NUM_WAITERS and NUM_SPINS were both set to 1. While the low contention performance is not as good as that of the test-and-set lock, fixing the position of the logical lock and eliminating the fetch-and-increment help in reducing the latency compared to that of the array lock. The latency of the new lock is 13% lower than that of the array lock for each single processor acquire-release operation. Both the *shared array* and the array lock have similar scalability. The *shared array* lock has slightly better performance numbers for all contention levels. The reason for this is that when the lock is released one of the processors spinning on the logical lock quickly enters the critical section while the others adjust their position in the array.

In summary, the advantages of the *shared array* lock are:

- Low latency and good scalability. This eliminates the high latency problem of queue-based schemes.
- Unlike other queue-based locking schemes, the *shared array* lock allows spin waiting processes to be preempted from their processor without blocking the other processors waiting to acquire the lock.

The new lock has a few limitations as well. They are:

- The problem of waiting on multiple critical sections is not solved by our scheme.
- The lock achieves its better latency but uses extra bus transactions. This could affect its scalability for large-scale multiprocessors.
- The lock does not guarantee FIFO behavior.

```
initialize()
// Initialize the lock array locations.
     lock[0] = UNLOCKED;
     for (i = 1; i < NUM_PROCS; i++)
          lock[i] = LOCKED;
     nextFreeLocation = 0;
acquire()
// Get a location to spin on.
// FAI is the atomic fetch and increment instruction.
     location = FAI(nextFreeLocation);
// Spin on the array location and await turn
// to enter critical section.
     while (lock[location mod NUM_PROCS]
          = LOCKED);
// Relock the position just vacated
// for future use by another process.
     lock[location mod NUM_PROCS] = LOCKED;
release()
// Release the next in line process if any.
     lock[(location + 1) mod NUM_PROCS]
          = UNLOCKED;
```

Figure 3: Anderson's Array Lock

4.4 Argument for Correctness

We present a proof that the *shared array* lock works correctly. A lock should provide mutually exclusive access to the critical section it protects and guarantees progress.

Theorem 1 *The shared array lock provides mutual exclusion.*

Proof: Any process that wants to enter the critical section has to execute the *acquire* procedure. There are exactly four ways a process can return from *acquire*. These returns occur at lines 2, 5, 17 and 29 of *acquire*. In each case the return is preceded by a successful test-and-set(TAS) on *lock*[0]. All that remains to be proved is that if TAS (*lock*[0]) is successful then the critical section is empty.

Assume TAS (*lock*[0]) is successful for a process p. TAS (*lock*[0]) is successful only when *lock*[0] has the value UNLOCKED. So before p did the successful TAS (*lock*[0]), some other process, say q, had to set the value of *lock*[0] to UNLOCKED. There are only two statements that set *lock*[0] to UNLOCKED. They are at line 4 during lock initialization and at line 1 of the lock *release* procedure. In the first case the

```
initialize()
// Waiters at array position 0 (lock[0]).
1       waiters = 0;
// waitersLock protects the waiters count
// - initialize it to UNLOCKED.
2       waitersLock = UNLOCKED;
// Initialize all the elements of the lock array
// to UNLOCKED.
3       for (i = 0; i < NUM_PROCS; i++)
4            lock[i] = UNLOCKED;

acquire()
// TAS is the test and set instruction.
1       if (TAS (lock[0]) == UNLOCKED)
2            return;
// If the current number of waiters is less than
// NUM_WAITERS, spin on lock[0].
// check&increment increments waiters and returns
// TRUE in this case and FALSE otherwise.
3       if (check&increment(waiters) == TRUE) {
4            while (TAS (lock[0]) == LOCKED);
5            return;
6       }
// If not permitted to spin on lock[0], search
// for a free position in the lock array.
7       index = 0;
8 find: do {
9            while (lock[++index] == LOCKED);
10      } while (TAS(lock[index]) == LOCKED)
// Spin on lock[index].
11      while (TRUE) {
12           for (s = 0; s < NUM_SPINS; s++) {
// If some other process frees the process,
// it advances one spot in the array.
// If it reaches lock[0] it joins the waiters there.
// Otherwise it repeats the search for a free position.
// It also frees the next waiter in line.
13                if (lock[index] == UNLOCKED) {
14                     if (--index == 0) {
15                          increment (waiters);
16                          while (TAS (lock[0])
                                 == LOCKED);
17                          return;
18                     }
19                     lock[index + 2] = UNLOCKED;
20                     index--;
21                     goto find;
22                }
23           }
```

Figure 5: The Shared Array Lock

```
// If no one frees the process and the process
// has spun NUM_SPIN times on lock[index],
// it checks lock[index - 1]. If lock[index - 1] is
// UNLOCKED the process frees its own position
// (lock[index]) and moves up in the array. Otherwise
// it starts spinning on its current position.
24        if (lock[index -1] == UNLOCKED) {
25            lock[index] = UNLOCKED;
26            if (--index == 0) {
27                increment (waiters);
28                while (TAS (lock[0]) == LOCKED);
29                return;
30            }
31            lock[index + 2] = UNLOCKED;
32            index--;
33            goto find;
34        }
35    } /* while */

release()
// Free the first two array positions
// and decrement the waiters count.
1     lock[0] = UNLOCKED;
2     lock[1] = UNLOCKED;
3     decrement(waiters);
```

Figure 6: The Shared Array Lock (continued)

critical section is empty because p is the first process ever to acquire the lock. Since TAS($lock$[0]) atomically sets $lock$[0] to LOCKED, no other process can enter the critical section till some process sets $lock$[0] to UNLOCKED. The only process that can do this is p as it is the only one which can execute line 1 of *release*. Thus for the first process p, the critical section is empty when it executes TAS($lock$[0]) and remains empty until it exits the critical section. By induction on the processes which entered the critical section since lock initialization, q is the only process in the critical section when it sets $lock$[0] to UNLOCKED at line 1 of *release*. So when p enters the critical section after the TAS($lock$[0]), it is the only process in the critical section. Using an argument similar to the one above, the critical section remains empty while p is in the critical section. Thus there is at most one process in the critical section at any instant.

Theorem 2 *The shared array lock ensures progress.*

```
check&increment(waiters)
// Check whether the number of waiters at lock[0]
// is less than the maximum allowed (NUM_WAITERS).
// If yes increment the waiters count and return
// TRUE, else return FALSE.
1     spin = FALSE;
2     while (TAS(waitersLock) == LOCKED);
3     if (waiters < NUM_WAITERS) {
4         waiters++;
5         spin = TRUE;
6     }
7     waitersLock = UNLOCKED;
8     return spin;

increment(waiters)
// Atomically increment the number of waiters.
1     while (TAS(waitersLock) == LOCKED);
2         waiters++;
3     waitersLock = UNLOCKED;

decrement(waiters)
// Atomically decrement the number of waiters.
1     while (TAS(waitersLock) == LOCKED);
2         waiters--;
3     if (waiters < 0)
4         waiters = 0;
5     waitersLock = UNLOCKED;
```

Figure 7: Shared Array Lock - auxiliary functions

Proof: Here we prove that the critical section remains empty only for a bounded time if there are processes waiting to enter it. We use the following lemmas in our proof.

Lemma 1 *lock[0] is never LOCKED unless there is a process in the critical section.*

Proof: The only places where $lock$[0] is set to LOCKED are in the TAS($lock$[0]) statements at lines 1, 4, 16 and 28 of *acquire*. In all four cases, the process executing the TAS immediately exits *acquire* and enters the critical section. Before exiting the critical section, all processes execute *release* in which $lock$[0] is set to UNLOCKED. Thus $lock$[0] remains LOCKED only when there is a process in the critical section. Thus the result.

Lemma 2 *lock[i], $i > 0$ is never LOCKED unless there is a process spinning on it.*

Proof: Initially $lock[i]$, $i > 0$ is set to UNLOCKED. The only way $lock[i]$ can be set to LOCKED is if a process does a successful test-and-set on it. This can happen in only in line 10 of *acquire*. Once a process does a successful test-and-set, it enters the infinite loop of line 11 and starts spinning on $lock[i]$. The process can only exit from the infinite loop at lines 17 or 21 (if $lock[index] ==$ UNLOCKED), and lines 29 or 33 (if $lock[index - 1] ==$ UNLOCKED). In the first case $lock[index]$ is already UNLOCKED. In the second, the process sets $lock[index]$ to UNLOCKED at line 25 before exiting at lines 29 or 33. Thus when a process stops spinning on $lock[i]$, $i > 0$, $lock[i]$ is always UNLOCKED. Thus the lemma.

Assume that the $lock[0]$ is UNLOCKED and, without loss of generality, array location i is the first LOCKED location in the array. Using the previous lemmas, there has to be a process, say p, spinning on this location. Process p checks position $i - 1$ once every NUM_SPINS spins on position i (line 24 of *acquire*). Since every location from 0 to $i - 1$ is UNLOCKED, it eventually reaches position 0 after $(i-1) * NUM\_SPINS$ spins on positions i through 1, and $i-2$ tests on positions $i-1$ through 0 in the worst case. After a final test-and-set on position 0, processor p enters the critical section. The *shared array* lock thus guarantees progress.

5 Conclusions

The experiments we conducted show that no current lock performs the best for all critical sections. If the expected contention for a critical section is known, an appropriate lock can be chosen using contention and critical section size as parameters. For critical sections with dynamically changing contention, this choice is difficult and is likely to be sub-optimal. The new *shared array* lock improves on the array lock in terms of latency and has comparable scalability. It alleviates the problem of processor preemption experienced by most queue-based locking schemes. The new lock is therefore a good choice for critical sections with dynamically varying or high contention.

References

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

[2] Encore Computer Corp. *Multimax Technical Summary*. Encore, Marlborough, Massachusetts, 1986.

[3] National Semiconductor Corporation. *Series 32000 Databook*. NSC, Santa Clara, California, 1986.

[4] Gary Graunke and Shreekant Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *IEEE Computer Magazine*, pages 60–69, June 1990.

[5] Maurice Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, 1991.

[6] Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

[7] Nayeem Islam and Roy Campbell. "Uniform Co-Scheduling using Object-Oriented Design Techniques". In *International Conference on Decentralized and Distributed Systems*, Palma de Mallorca, Spain, September 1993.

[8] Henry Massalin and Calton Pu. A Lock-Free Multiprocessor OS Kernel. Technical Report CUCS-005-91, Columbia University, 1991.

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

[10] Bodhisattwa Mukherjee and Karsten Schwan. Experiments with Configurable Locks for Multiprocessors. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages II–205–II–208, Chicago, Illinois, August 1993.

[11] Vincent F. Russo. *An Object-Oriented Operating System*. PhD thesis, University of Illinois at Urbana-Champaign, January 1991.

[12] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared–Memory. Technical Report CSL-TR-91-469, Stanford University, 1991.

[13] Josep Torrellas, Anoop Gupta, and John Hennessy. Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System. In *ASPLOS V Proceedings*, pages 162–174, October 1992.