Composite Abortable Locks

Virendra J. Marathe[1], Mark Moir[2], and Nir Shavit[2]

[1]University of Rochester
Department of Computer Science
Rochester, NY 14627, USA
vmarathe@cs.rochester.edu

[2]Sun Microsystems Laboratories
Burlington, MA 01803, USA
{mark.moir, nir.shavit}@sun.com

Abstract

The need to allow threads to abort an attempt to acquire a lock (sometimes called a timeout) is an interesting new requirement driven by state-of-the-art database applications with soft real-time constraints. This paper presents a new composite abortable lock (CAL), a combination of abortable queue-based (QL) and test-and-set based backoff (BL) lock mechanisms, which provides non-blocking aborts while ensuring low space requirements without need for a memory reclamation scheme. The key observation motivating our approach is that the fast lock hand-off achieved by QLs only requires the first few threads to be queued (not all *waiting threads), and that the remaining threads can run as in a BL. We developed an algorithm that uses only a short fixed size structure for queueing, allowing most threads to back-off. This reduces worst-case space overhead dramatically, and improves performance by eliminating the need for expensive and complicated memory management mechanisms.*

Experimental results show that our new CAL algorithm not only saves on space, it actually outperforms Scott's state-of-the-art nonblocking abortable QL under contention, and even more so when there are more threads than processors. Moreover, as the rate of lock aborts increases, the CAL continues to perform well, while Scott's algorithm deteriorates rapidly.

1. Introduction

Some parallel applications require threads to be able to abandon an attempt to acquire a lock if it takes too long, for example to break deadlocks and to meet soft real-time requirements [10]. In attempting to acquire a lock that supports such functionality, a thread specifies a *patience* value, indicating how long it is willing to wait to acquire the lock. If the thread does not acquire the lock in the specified time, it returns with an indication that it failed to acquire the lock. Locks supporting this functionality are sometimes called *try locks* [10]; we call them *abortable locks*.

While lock abortability is easy to support with traditional back-off locks (BLs), these locks do not scale well under heavy contention. The state-of-the-art scalable abortable lock for cache-coherent multiprocessors is the *nonblocking abortable CLH Queue-lock* due to Scott [9]. Though it significantly outperforms BLs under contention, Scott's algorithm suffers from various drawbacks compared to BLs: it has a high space overhead, it requires specialized memory management, and its performance is inferior under preemption.

1.1. Composite Abortable Locks

This paper presents a new scalable *composite abortable lock* (CAL) which provides non-blocking aborts while ensuring low space requirements, and without need for a memory reclamation scheme. The new CAL algorithm is based on the observation that the fast lock hand-off achieved by QLs only requires the first few threads to be queued, not *all* waiting threads. The remaining threads can simply back-off. Based on this observation, we build a lock that combines the key algorithmic features of the BL and QL algorithms.

In a nutshell, our CAL algorithm works as follows. We keep a small fixed size array of lock nodes (in our benchmarks on a 30 processor machine we used an array of size 4). Each thread accessing the lock selects a node to use in its acquisition attempt. If the node is in use by another thread, the thread backs off exponentially and retries, either on the same node or a different one selected at random. If the thread succeeds it uses the node in a "mini" CLH style queue-lock algorithm based on a linked list implemented in the array.

1-4244-0054-6/06/$20.00

The thread spins on the node preceding it in the list, and when the thread owning the preceding node signals that it has finished its critical section, the thread accesses the critical section. It releases its node in-turn when it completes. If a thread needs to abort while owning a node, it leaves it after a minor update of the node state. If it needs to abort while backing-off, it simply leaves.

This mechanism provides several interesting algorithmic properties. It distributes backed-off access attempts onto several locations, reducing contention. It provides the scalable quick-handoff mechanism of CLH queue locks for the nodes on the lock's critical access path. It provides nonblocking aborts without the need for memory management since all aborts that occur while a thread is backed-off have zero-cost as in BLs, and for ones that happen while holding a node, one can simply leave the array node around since some other thread will select it at random for future use. For L locks and T threads, the new algorithm requires only $O(L)$ space in the worst case, as compared with $O(L * T)$ for Scott's abortable QL algorithms. Given all these advantages, one needs to keep in mind that the new algorithm does not provide the same FIFO lock access fairness as non-abortable QL. Scott's algorithms do.

1.2. Performance

We ran a set of experiments on a 30-processor Sun Enterprise 6000, a cache-coherent NUMA machine. Our new CAL not only scales as well as Scott's algorithm, but also noticeably outperforms it. This scalability verifies the key insight that only the front part of the QL is necessary to ensure fast lock handoff in a scalable way. As we increase the number of threads beyond the number of processors, the percentage of failed (timed-out) acquisition attempts increases only modestly for BL and CAL algorithms, yet the degradation is severe with Scott's algorithm. This is because its queue is significantly longer than that of the CAL, and preemption within the queue has a high performance cost. Finally, when we measure the number of failed acquisition attempts as the patience value decreases, we see that our new CAL degrades gracefully while Scott's algorithm deteriorates rapidly.

1.3. Related Work

Traditional *test-and-set* (TAS) locks represent lock ownership in a single word, and a thread wishing to acquire the lock repeatedly attempts to atomically change the lock from "free" to "owned" (for example) using an atomic synchronization instruction such as `test-and-set` or `compare-and-swap` (CAS). Designing an abortable version of such locks is trivial: If a thread exceeds its patience without acquiring the lock, it simply gives up and returns.

However, TAS locks are known to perform poorly under heavy load because of the memory traffic caused by the repeated atomic instructions on the same memory location [1]. TAS locks can be improved somewhat by reading the lock word first, and attempting to acquire it only if it is not held, yielding so-called test-and-test-and-set (TATAS) locks. TAS and TATAS locks can be further improved by using *backoff*: When a thread fails to acquire the lock, it delays for some time before trying again, thereby reducing contention for the lock. While these techniques help somewhat, TAS and TATAS locks are still not scalable. Furthermore, it is difficult to make threads backoff just the right amount, and as a result, handing the lock off from one thread to another can take significantly longer than necessary, resulting in limited throughput.

TAS and TATAS locks have the additional disadvantage of exhibiting very unfair behavior because they impose no ordering whatsoever between threads attempting to acquire the lock. Because a thread that has just released the lock has it cached, it can often acquire the lock again without allowing other threads an opportunity to do so. While this can appear to produce high throughput, it actually starves other threads for long periods of time, which is undesirable in many applications.

The shortcomings of these simple locks have lead to extensive research into scalable *queue locks* (QLs) [1, 2, 3, 7, 8], which generally arrange for threads to form a queue so that each thread "spins" on a different memory location, allowing those locations to be cached until the spinning thread is informed by its predecessor in the queue that it can acquire the lock. Most of these locks do not allow threads to abort, however.

Scott and Scherer [10] proposed abortable versions of two classic queue-locks: the MCS lock [8] and the CLH lock [2, 7]. However, these locks require an aborting thread to wait for another thread, undermining the value of being able to abort to meet soft real-time requirements. Scott [9] subsequently proposed *non-blocking* versions of these locks to overcome this problem. Although these nonblocking abortable QLs are as scalable as the original QLs, their worst-case space requirements are unbounded. Scott does suggest an approach for limiting memory usage to $O(L * T)$ for L locks, where T is the number of threads in the system, but he does not consider this approach worthwhile in practice. He suggests that the space overhead

may be a theoretically unavoidable limitation of the non-blocking abortability property of queue-locks.

Jayanti [5] proposed another abortable queue lock, but this lock has the severe practical disadvantages of *best-case* space overhead that is linear in the number of threads that *might* access the lock, as well as the need to know an upper bound on the number of such threads.

Lim and Agarwal [6] suggested the *reactive lock*, which switches among several lock types, viz. QL and BL, based on the concurrency/load level on the lock. They did not show how to allow aborts. While their scheme can probably be extended to allow aborts, by using abortable QLs (e.g., [5, 9]) in place of regular ones, such a scheme would inherit the disadvantages of these QLs already discussed, including high space overhead and complicated memory management mechanisms. The composite lock approach carefully combines the advantages of BLs and QLs, while avoiding their disadvantages, rather than attempting to switch among different schemes.

The remainder of this paper is organized as follows. In Section 2, we present a high-level overview of our CAL approach, and in Section 3, we present a specific example of a CAL lock based on the well-known CLH lock, and discuss some important optimizations. We present performance experiments in Section 4. Concluding remarks appear in Section 5.

2. Overview

The key insight underlying our Composite Abortable Lock (CAL) approach is that

1. Only the front part of the QL needs to exist in order to provide the tightly timed behavior of the algorithm. All other threads can exponentially backoff but need not be in a tight list.
2. Distributing the backoff mechnism onto multiple locations will reduce effects of contention incurred by the single location in the BL.

Thus in a sense we combine the best features of QLs and BLs, namely tight coordination at the front of the queue and low space overhead. A number of locks may be implemented by the Composite Lock approach. In this section, we provide a brief high-level description of such locks, and in the next section we present one specific implementation in detail, and describe an optimization to it.

In most QLs, each thread attempting to acquire the lock allocates a *queue node*, which it inserts into a FIFO queue. Having done so, the thread then spins on a node in the queue—usually its own node or its predecessor in the queue—until a change to this node indicates that it can now enter its critical section. The need for one node per thread accounts for the space overhead of QLs, and also necessitates some form of memory management for these nodes, which is more complicated in algorithms that support aborts.

Rather than allocating a node per thread, and using memory management techniques to reclaim nodes after use, CAL algorithms based on our approach use a small array of C nodes for some constant C. These nodes are used to in a manner similar to the nodes in existing QL algorithms to create an order between waiting threads, but because we only aim to tightly coordinate a few threads at the front of the queue, we do not need one node per thread.

The basic structure of algorithms based on our CAL approach is that, rather than *allocating* a node before beginning, a thread *acquires* one of the C pre-allocated nodes in the array. Having acquired a node, a thread uses it in a queueing algorithm similar to those in the literature. Threads waiting to acquire a node can backoff in a variety of different ways to reduce contention and memory traffic. Because the threads that have acquired nodes are tightly coordinated in the queue, this backing off does not affect the time it takes for a thread to handoff the lock to another thread, as is the case in pure backoff locks.

The number C of nodes needed for the array will depend on a variety of factors including architecture, application behavior, etc. While we present the specific algorithm in the next section in terms of a fixed-sized array, it is not hard to grow the array dynamically if necessary.

CAL implementations may employ a variety of techniques for selecting a node to acquire. For example, this selection may be completely random (as in the concrete CAL algorithm presented in the next section), systematic (where threads use specific nodes that are assigned to clusters of threads based on some system-specific heuristic), or a combination of both (for adapting to the runtime workload; for example, threads usually backoff on a fixed subset of nodes, but occassionally backoff on randomly selected nodes if the load on their default node subset increases above a particular threshold). If the chosen slot is already owned by another thread, the current thread may choose to backoff for some time before retrying the slot acquisition step, and it may try again to acquire the same node, choose a different node, etc. Additional heuristics to make a thread backoff and retry on multiple nodes (that are selected randomly or in a specific order) may also be used to increase the chances of early node acquisition

by the thread.

Similarly, a variety of backoff strategies can be employed by threads waiting to acquire a node.

Supporting aborts in CALs is in general much simpler than in previous abortable QL algorithms. First, if a thread decides to abort its attempt to acquire the lock before acquiring a node in the array, no further action is necessary; it can simply leave, just as in BLs. Presumably aborts are most common under heavy contention, and so this is the most common case. Even for a thread that does acquire a node before deciding to abort, aborting is generally significantly simpler in CALs based on our approach than in previous abortable QLs. This is because, by using a fixed array of nodes, we avoid the complicated memory management techniques of previous abortable QLs, and thus an aborting thread does not need to interact with such mechanisms. For example, in the specific CAL algorithm presented in the next section, a thread that decides to abort after acquiring a node performs two simple stores before leaving.

We present a composite abortable lock based in the CLH lock [2, 7] in the next section.

3. A CLH-based CAL

The example Composite Abortable Lock we present in this section is based on the CLH lock [2, 7]. Briefly, the CLH lock works as follows. To acquire the lock, a thread first allocates a node, sets the node's status to W (for "wait") and inserts it into a wait-queue. If the node is the first in the queue, the thread immediately enters its critical section. Otherwise, it spin-waits on the status of its predecessor node, waiting for it to become R (for "release"). When a thread completes its critical section, it sets the status of its node to R, thereby releasing the next thread in the queue (if any) into its critical section. The order of nodes in the wait-queue can be represented *explicitly* by having each thread store the address of its predecessor's node in its own node, or *implicitly* by having each thread remember its predecessor in a private variable. In the CLH-based CAL presented below, a thread records its predecessor implicitly in a private variable until such time as it aborts, in which case it stores the predecessor explicitly in its node, allowing other threads to clean up its aborted node. This approach avoids the shared memory store in the common case, where threads do not abort their lock acquisition attempt.

We now describe our CLH-based CAL in more detail. The data types and initialization are shown in Figure 1, along with pseudocode for the `release` method, and pseudocode for the `acquire` method is shown in Figure 2. Each lock L consists of a constant size (C) node array *buffer[1..C]* (Lines 1 to 4). Each node consists of a `state` field and a pointer to the `next` node in the CLH wait-queue (Lines 9 to 13). The queue's `Tail` (Lines 5 to 8) consists of a pointer to the tail node of the wait-queue, and a version number of the `Tail`. A version number is required to avoid the *ABA* problem [1]. At any point in time, the `Tail` pointer is either `NULL` or a pointer to the last node inserted in the wait-queue. A node may be in any of the four following states (Lines 10 and 11): *waiting (W)*, *released lock (R)*, *aborted (A)*, or *free (F)*. We now describe the `acquire` method that a thread uses to acquire the lock. We begin by describing the general structure of this method without regard to abort functionality, which is described later.

A node is in state F initially. A thread intending to acquire the lock randomly selects a node for use (Line 18) and if the node is in state F, atomically switches it to state W, using a CAS (Line 23). If however, the node is not in state F, the thread retries (the loop from Line 22 to 43) with exponential backoff (Line 39) between failed attempts, until its atomic acquisition (Line 23) succeeds. Essentially, the thread does a TAS lock with backoff on the node.

After a thread has successfully acquired a node (Line 44 onwards), which is now in state W, it attempts to atomically insert the node at the wait-queue's `Tail` (Lines 44 to 51). Then, if the inserted node is the first in the queue, the thread has successfully acquired the lock (Lines 53 and 54). If not, the thread spin-waits on the next node in the queue until the node switches to state R (Lines 56 to 68). The thread then sets the state of the node on which it was spinning to F (Line 69), allowing the node to be reused later. After executing its critical section, the thread releases the lock by updating its node's state to R (Line 14), thereby informing the next thread in the queue, if any, that it can enter the critical section.

[1] The *ABA* problem is a side effect of atomic `compare-and-swap` (CAS) instruction's semantics. `CAS(addr,old,new)` atomically checks to see if the value at address `addr` is the same as `old`, and if so, swaps `new` into `addr`. Hence, if thread $T1$ reads a value, say A, from location L, and before it does a `CAS(L,A,C)`, thread T2 modifies L to a new value, say B, and later writes back A into L, thread $T1$'s CAS will succeed even though thread $T2$ modified L several times between the read and CAS of L by $T1$. Because the intention behind such uses of CAS is usually that the CAS should succeed only if the variable accessed does not change between the load and the CAS, this often leads to incorrect behavior. By augmenting the variable with a version number that is incremented on every update, we avoid this problem in practice.

3.1. Abort Capability

As mentioned earlier, our lock supports timeout capability wherein a thread may abort waiting for a lock after it runs out of patience. If the thread is in the backing off stage (i.e. it has not acquired a node yet) then it can abort its attempt without informing other threads because it has not yet entered the queue (Lines 41 and 42). If the thread decides to abort when already in the wait-queue, to preserve a consistent queue structure, it makes the next node in the wait-queue (that the thread is spinning on) explicit (Line 63); the thread then sets its node's state to A (using a memory store instruction, Line 64), effectively aborting its attempt of acquiring the lock, and leaves. In this case, the thread, if any, that owns the node immediately behind this recently aborted node in the wait-queue is responsible for cleaning up that aborted node (Lines 57 to 61). The thread does so by changing its predecessor to the aborted node's predecessor (Line 59), and then setting the aborted node's state to F, preparing it for future reuse (Line 60). Both updates can be done with simple memory stores. The thread subsequently spin-waits on the *new* predecessor node. If the new predecessor is also in state A, that node is also cleaned up in the same fashion. As described below, if there is no node behind a recently aborted node (i.e. the aborted node is the `Tail`), then the aborted node can be cleaned up by a thread that subsequently inserts its acquired node at the `Tail`, behind the aborted node.

According to the algorithm as presented thus far, if a thread intends to acquire a node that is in either state A or state R, the thread waits for the node to be cleaned up by another thread. However, if the node is the wait-queue's `Tail`, no other thread will clean it up. We require a different mechanism to handle this case. The mechanism we present in our algorithm (Line 27 to 38) allows the thread intending to acquire a node to clean up that node if the node is the current `Tail` of the wait-queue and to simultaneously acquire the node. To ensure that only one thread successfully acquires the node, this is achieved by changing the `Tail` using CAS, so that it no longer points to the node. Having thus acquired the node, the thread then stores W in the node's state field (Line 34), and then proceeds as usual to link the node into the queue. The new value stored in the `Tail` pointer during this special-case node acquistion depends on the state of the node:

State R: If the state of the acquired node was R, then the node was necessarily the *only* node in the wait-queue (since the node must become the head of the queue before switching to state R). In this case, the new value of the `Tail` pointer is NULL (Line 30 to 33), indicating that the wait-queue is now empty. The next thread that inserts its node into the wait-queue (perhaps the same one that set the `Tail` pointer to NULL) will have implicitly acquired the lock (Line 52 to 54), since there exists no predecessor node in the queue.

State A: Because the first thread to link a node into the queue after the queue is empty immediately enters the critical section, it does not abort. Furthermore, when it leaves the critical section, it sets its node to state R. Therefore, whenever the queue is not empty, its first node is always in state W or state R. This implies that, if the state of the node acquired during the above-described tail cleanup was A, then the node was not at the head node of the wait-queue; i.e., there must exist at least one node ahead of that node in the wait-queue. Therefore, in this case, the new `Tail` pointer stored by the cleanup is the predecessor of the node being cleaned up and acquired (Line 30 to 32).

3.2. An Important Optimization

We designed the algorithm described above with performance and scalability under heavy contention in mind. However, performance in the absence of contention is also important. Ideally the lock should be almost as simple and streamlined as a simple TAS lock in this case. In the algorithm as presented above, we will often have to move the `Tail` pointer off a released node, claim the node, and then splice it into the queue in the uncontended case. In this section, we briefly describe an optimization that aims to make the lock behave very similarly to a simple TAS lock in the absence of contention.

The key idea behind our optimization is to allow a thread to acquire the lock without acquiring a node and splicing it into the queue if there is no other thread holding or requesting the lock. We achieve this by adding a `locked_by_unqueued_thread` bit to the `Tail` variable (which still has a node pointer and a version number as before). A thread that finds the pointer to be NULL and the `locked_by_unqueued_thread` bit false can acquire the lock by simply using CAS to set the bit to true (while ensuring that the pointer is still NULL). Thus, an uncontended acquire in this optimized algorithm is almost the same as a simple TAS lock.

The main challenge involved in making this optimization work is making it compatible with queueing threads as before under contention. We also need to address how we get back to the simple, uncontended mode after a period of high contention.

We considered a number of optimization approaches, and found several that appeared to be workable. In some of our optimizations, we went to great lengths to keep uncontended behavior as faithful as possible to a simple TAS lock. In particular, we worked hard to ensure that a thread that acquires the lock in the absence of contention can release the lock using a simple store, as opposed to a more expensive CAS operation. While we did achieve such an algorithm, it was significantly complicated by our efforts to avoid using CAS for an uncontended release. Furthermore, we found that using CAS for an uncontended release did not harm performance significantly. We therefore decided on a simpler optimization, which is described below.

If a thread wishing to acquire the lock finds the `locked_by_unqueued_thread` bit set, then there is contention for the lock, so the thread acquires a node and splices it into the queue, largely in the same way as in the algorithm described above. However, because of the thread holding the lock without getting in the queue, the thread cannot infer that it has acquired the lock simply because it is first in the queue, as before. Instead, the first thread in the queue must wait until the `locked_by_unqueued_thread` bit is set to false by the unqueued thread. The race between the unqueued thread releasing the lock and a new thread splicing its node into the queue explains why the unqueued thread must use a CAS to release the lock.

Threads that do join the queue behave largely the same in the optimized algorithm as they do in the unoptimized one described previously except that, as described above, a thread at the front of the queue cannot acquire the lock until the `locked_by_unqueued_thread` bit is false. Such operations never modify the `locked_by_unqueued_thread` bit; this is done only by a thread that acquires the lock without joining the queue.

After a period of contention in which the queue is used, there are a number of safe ways to reset the `Tail` to NULL so that the lock again behaves like a TAS lock until contention arises again. In our implementation, this is the responsibility of the next thread to acquire the lock. This thread observes that the `Tail` pointer points to a node in R state, or a node in A state with a NULL next field, indicating that the queue is empty. In this case, the thread uses CAS to set the `Tail` pointer to NULL and the `locked_by_unqueued_thread` bit to true, thereby acquiring the lock, and restoring it into uncontended mode.

We examine the performance of our algorithm, and the effect of the optimization just described, through a series of experiments in the next section.

```
1:  typedef struct {
2:     TailNode Tail;          // contains pointer to tail node
                               // and a version number
3:     QNode buffer[C];        // C is a constant
4:  } Lock;

5:  typedef struct {
6:     QNode *pNode;
7:     unsigned int version;
8:  } TailNode;

9:  typedef struct {
10:    unsigned int state;     // free (F), waiting (W),
11:                            // released (R), or aborted (A)
12:    QNode *next;
13: } QNode;

initially:
    L->Tail = <NULL,0>
    for all i = 1..C: L->buffer[i].state = F

void release(Lock *L, QNode* acq_node)
14: acq_node->state = R;
```

Figure 1. Data types, initialization, and code for release method.

4. Experimental Results

In this section we present results of the experiments we conducted to compare our CALs with the TAS-Backoff lock and the CLH-based QL with nonblocking aborts [9]. Our experiments show that our CAL scheme is as scalable as this CLH QL and consistently outperforms it under contention. They also show that our algorithm degrades significantly more gracefully than the CLH-based QL under adverse conditions such as preemption and small patience values.

We performed our experiments on a 30-processor Sun Enterprise 6000, a cache-coherent NUMA machine formed from 15 boards each with two 366MHz UltraSPARC® II processors and 2GB of RAM. The C code was compiled with a Sun *cc* compiler 5.3, with the flags `-xO5` and `-xarch=v8plusa`. Each plotted data point represents an average over three executions in which each thread performs 100,000 lock acquisition attempts.

In order to test the locks under heavy contention, we created an artificial microbenchmark in which each thread repeatedly attempts to acquire the lock and executes a critical section if it succeeds. The critical section is a tight 300 nanosecond loop, where a thread accesses only its processor's high resolution timer. The non-critical work (between consecutive attempts to execute the critical section) also consists of a 300 nanosec-

```
QNode* acquire(Lock *L, unsigned long patience)
15: unsigned long start_time = CurrNanoTime();
16: int nano_backoff = INIT_BACKOFF;
17: unsigned int tail_version;
18: int index = ChooseRandomInt(0,C-1);
19: QNode *acq_node = &L->buffer[index];
20: QNode *implicit_next = NULL;
21: TailNode currTail;
// acquire a node
22: while(TRUE) {
23:    if(CAS(&acq_node->state,F,W))
24:       break;
25:    currTail = L->Tail;
26:    int node_state = acq_node->state;
27:    if((node_state == A) || (node_state == R)) {
28:       if(acq_node == currTail.pNode) {
29:          QNode *next_node = NULL;
30:          if(node_state == A)
31:             next_node = acq_node->next;
32:          if(CAS(&L->Tail,currTail,<next_node,
33:                 currTail.version+1>)) {
34:             acq_node->state = W;
35:             break;
36:          }
37:       }
38:    }
39:    backoff(ChooseRandomInt(0,nano_backoff));
40:    nano_backoff <<= 1;
41:    if((CurrNanoTime() - start_time) > patience)
42:       return NULL;
43: }
// splice node into queue
44: do {
45:    currTail = L->Tail;
46:    if((CurrNanoTime() - start_time) > patience) {
47:       acq_node->state = F;
48:       return NULL;
49:    }
50: } while(!CAS(&L->Tail,currTail,
51:              <acq_node,currTail.version+1>));
// wait for predecessor to release lock
52: QNode *implicit_next = currTail.pNode;
53: if(implicit_next == NULL)
54:    return acq_node;
55: int next_state = implicit_next->state;
56: while(next_state != R) {
57:    if(next_state == A) {
58:       QNode *tmp = implicit_next;
59:       implicit_next = implicit_next->next;
60:       tmp->state = F;
61:    }
62:    if((CurrNanoTime() - start_time) > patience) {
63:       acq_node->next = implicit_next;
64:       acq_node->state = A;
65:       return NULL;
66:    }
67:    next_state = implicit_next->state;
68: }
69: implicit_next->state = F;
70: return acq_node;
```

Figure 2. Code for acquire method.

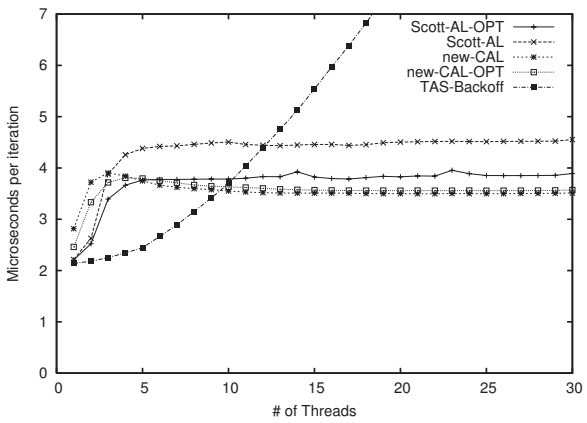

Figure 3. Performance and Scalability: With a large patience value, and varying the number of threads from 1 to 30 we find that our CALs are as scalable as QLs.

ond loop. In all of our experiments, we used a node array of size 4.

4.1. Performance and Scalability

In our first experiment, we vary the number of active threads between 1 and 30, and each thread attempts to acquire the lock specifying a very large patience value. Thus, this experiment tests the performance and scalability of the locks in the absence of aborts. The results are shown in Figure 3. The curve labelled "Scott-AL" is the algorithm published in [9], and the one labelled "Scott-AL-OPT" is the same algorithm with an optimization we applied to it (described below). Similarly, we use "new-CAL" and "new-CAL-OPT" for our CAL algorithm and the optimized version of it, respectively. "TAS-Backoff" represents the TAS lock with backoff.

We first consider the performance of the algorithms when only one thread is active, so there is no contention for the lock. The TAS-Backoff lock performs best in this case, as expected. The Scott-AL and Scott-AL-OPT algorithms perform almost as well, and new-CAL performs significantly worse. We can see that the optimization described in Section 3.2 indeed improves the single-threaded performance (new-CAL-OPT), but it still does not perform as well as the Scott-AL algorithms. Because the critical path through lock acquisition and release is so short in the simple uncontended case, optimizations at a lower level than we have implemented so far become important. We believe we can close the gap between new-CAL-OPT and the TAS-

Backoff and Scott-AL locks significantly, and plan to report this in the final version of the paper.

Next we consider scalability with increasing numbers of threads. We observe that TAS-Backoff performs increasingly poorly as contention increases, consistent with numerous previous studies, e.g. [9]. We see that Scott-AL scales well with increasing contention, also consistent with [9].

In the `release()` method of Scott-AL, a thread performs a CAS to attempt to change the `Tail` to NULL if it still points to the thread's node. However, this is usually not the case under contention because other threads have joined the queue, so the CAS is usually wasted. We applied a common optimization whereby the thread reads the variable and avoids an expensive CAS operation in case it would fail anyway. We see from Figure 3 that the optimization improves the performance of Scott-AL significantly.

Our CALs not only scale as well as Scott-AL and Scott-AL-OPT, but also noticably outperform them. The scalability verifies the key insight that only the front part of the QL is necessary to ensure fast lock handoff in a scalable way.

Our CALs perform better than Scott-AL and Scott-AL-OPT under contention because of our approach of avoiding the need to reclaim queue nodes by having a small fixed number of them. In the `release()` method of our algorithm, a thread hands off the lock to the next queued thread *and* frees the node for subsequent reuse simply by storing R to its node's status field. In contrast, the Scott-AL and Scott-AL-OPT algorithms require two separate actions: one to check whether the `Tail` still points to its node (which it generally does not under contention), and one to release the lock to the next thread. These actions occur on two different cache lines, one of which (the `Tail` pointer access) is unlikely to be cached in the presence of contention, whereas our `release()` method requires only a single cache line access which will typically be a cache hit.

4.2. Effect of Preemption

Our next experiment studies the effect of preemption on the various locks by using more threads than there are processors in the system. We repeat the same experiment as before, but now we vary the number of threads from 30 to 40, we use a patience of 512 microseconds, and we measure the percentage of lock acquisition attempts that fail due to timeout. The results are shown in Figure 4.

With 30 threads (i.e., exactly one per processor), all of the QLs have almost zero failures, while TAS-Backoff has almost 30% failures; because of the poor performance due to excessive contention, many lock acquisition attempts timeout before succeeding.

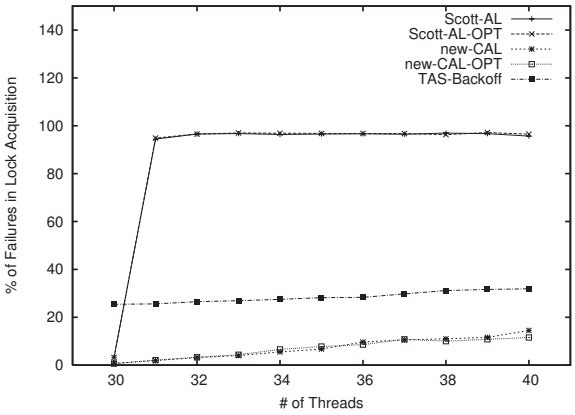

Figure 4. Preemption Tolerance: Preemption has a severe effect on the success rate for the Scott-AL algorithms because when a thread is preempted in the queue, all others behind it in the queue must wait, and are likely to be preempted themselves.

As we increase the number of threads, the percentage of failed acquisition attempts increases only modestly for TAS-Backoff and both of the new-CAL algorithms. Some degradation is to be expected, because occasionally a thread is preempted while holding the lock, and all others must wait until the preempted thread is rescheduled, resulting in excessive waiting leading to timeout.

However, the degradation is severe for both Scott-AL algorithms. The reason is that these algorithms queue *all* waiting threads. Note that if a thread is preempted while in the queue, all threads behind it in the queue must wait for it to be rescheduled before they can acquire the lock. While they are waiting, they too may be preempted, and a pathological situation results in which almost no acquisition attempts succeed. More recently, He et. al. have extended Scott's algorithms for preemption adaptivity [4].

In contrast, because our new-CAL algorithms queue only a small number of threads (at most 4 in our experiments), this effect is much less pronounced: threads spend less time in the queue, and are therefore less likely to be preempted while in the queue. Note that the preemption of a thread that has not yet acquired a node has no effect on other threads.

4.3. Degradation

In our last experiment, we ran 30 threads, and measured the percentage of failed acquisition attempts as we reduced the patience of the threads. The results are shown in Figure 5.

We first observe that at the highest patience value in this test (144 microseconds), the TAS-Backoff lock has about 60% failures while the Scott-AL algorithms achieve almost no failures. This is explained by a phenomenon previously observed in TAS-Backoff locks [9]. These locks are very *unfair* in that there is no reason why one thread cannot acquire the lock repeatedly while another fails to acquire it. In fact, because releasing the lock requires exclusive ownership of the cache line containing the lock word, the thread that has just released the lock is very likely to succeed in acquiring it again. While this happens repeatedly, other threads are kept waiting, and eventually they time out, which explains the high failure rate for TAS-Backoff.

In contrast, because the Scott-AL algorithms queue all threads attempting to acquire the lock, they are very fair: at worst a thread has to wait for every other thread to acquire and release the lock once. Thus, provided the patience value is large enough to accommodate one lock acquisition and release, no timeouts occur.

The new-CAL algorithms do not queue all waiting threads, so—as with TAS-Backoff—it is possible for one thread to acquire the lock multiple times while another thread repeatedly fails to do so. However, even the short queue that forms in this algorithm is sufficient to avoid the pathological scenario described above for TAS-Backoff, in which unfairness is almost guaranteed. Therefore, with patience of 144 microseconds, we observe the new-CAL algorithms suffering some failures, but much fewer than TAS-Backoff.

We next examine the behavior of the algorithms as we gradually reduce the patience value. For the TAS-Backoff and our new-CAL algorithms, the failure rate gradually increases as we reduce the patience. This is because all aborts for TAS-Backoff and almost all aborts for the new-CAL algorithms (i.e., those that occur before the thread has acquired a queue node) have no further effect on the performance of the other threads.

In contrast, because the Scott-AL algorithms immediately queue every thread, all aborts happen while the aborted thread has a node in the queue. When a thread aborts while it is in the queue, this impacts the performance of other threads, which are left with the responsibility of cleaning up and deallocating the aborted thread's node. This additional work makes

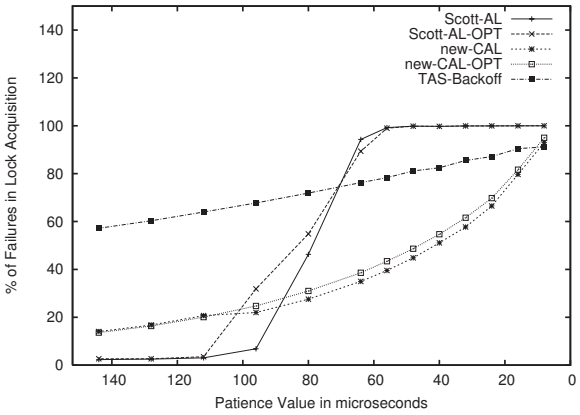

Figure 5. Degradation Test: The QL algorithms degrade suddenly as patience value decreases, while TAS-Backoff and our new CAL algorithms degrade gracefully. This is because aborts in TAS-Backoff and most aborts in our algorithm do not have any effect on the performance of other threads, while in the QL algorithms an abort requires other threads to do additional work, therefore becoming more likely to timeout themselves.

that thread wait longer, and therefore it is more likely to abort itself. Thus we have a "cascading" effect, in which aborts cause more aborts. This effect is clearly observed in Figure 5, where the Scott-AL algorithms are observed to quickly approach 100% failure when the patience value becomes small enough that aborts begin to occur.

These results show a clear advantage of our new CAL locks over the Scott-AL algorithms, because their success rates degrade gracefully as the load becomes large enough that some threads start aborting.

5. Concluding Remarks

We have proposed a new form of abortable mutual exclusion lock, which we call a Composite Abortable Lock. This lock shows significant performance benefits over known abortable locks. The key insight behind our approach is that, in order to achieve scalable performance under contention, we must tightly coordinate a few threads to allow a fast handoff using local spinning techniques, but that it is not necessary to tightly coordinate all waiting threads, as previous scalable locks do. Based on this observation, we show that it is pos-

sible to achieve the required coordination by queueing only a small number of threads, allowing others to simply backoff as in simple, nonscalable locks. We also show how to use a small, fixed structure to form this queue, eliminating the need for complicated and expensive memory management techniques, and also dramatically reducing worst-case space consumption.

References

[1] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[2] T. S. Craig. Building fifo and priority-queuing spin locks from atomic swap. Technical Report TR 93-02-02, Department of Computer Science, University of Washington, Feb. 1993.

[3] G. Granunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *Computer*, 23(6):60–69, 1990.

[4] B. He, W. N. Scherer III, and M. L. Scott. Preemption adaptivity in time-published queue-based spin locks. In *Proceedings of the 11th International Conference on High Performance Computing*, Dec. 2005.

[5] P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*, pages 295–304, 2003.

[6] B.-H. Lim and A. Agarwal. Reactive synchronization algorithms for multiprocessors. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 25–35, 1994.

[7] P. S. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 165–171, 1994.

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

[9] M. L. Scott. Non-blocking timeout in scalable queue-based spin locks. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, pages 31–40, 2002.

[10] M. L. Scott and W. N. Scherer III. Scalable queue-based spin locks with timeout. *Proceedings of the 8th Symposium on Principles and Practice of Parallel Programming*, 36(7):44–52, 2001.