Recovering Scalable Spin Locks

Philip Bohannon* Daniel Lieuwen Avi Silberschatz
Bell Laboratories

Murray Hill, NJ 07974
{plbohannon,lieuwen,avi}@bell-labs.com

December 15, 1997

Abstract

We present a mechanism for making a scalable spin lock protocol, the MCS lock, *recoverable*, thereby ensuring that a lock never becomes permanently unavailable, even if one or more processes using the lock die. This is achieved by modifying the original protocol to write additional information to shared memory and introducing a cleanup process which returns locks to a usable state in case of process death(s). Our method does not require kernel or hardware support other than the `swap` instruction, and maintains performance comparable to the original protocol (one third as fast in the uncontested case). We have proven the correctness of our scheme in the face of the weak memory models provided by modern systems.

Appeared in: Proceedings of the Eigth IEEE Symposium on Parallel and Distributed Systems, October 23-26, 1996, New Orleans, Louisiana.

*A Ph.D. candidate at Rutgers University.

1 Introduction

In shared memory computing environments, multiple processes running on one or more processors require synchronized access to shared structures. In an environment where performance is critical, a common method of implementing mutual exclusion is to directly use a hardware provided abstraction such as test-and-set or compare-and-swap, without involving the OS. An implementation in which a process repeatedly tries to acquire such a lock in a tight loop is called a *spin lock* (see, for example, [And90]). The activity of retrying is known as *busy waiting* or simply *spinning*.

In simple spin lock implementations, all processors on a multi-processor system will frequently access and attempt to write the lock control variable. On most modern computer architectures, each processor will also be attempting to *cache* the control variable. Since each update (or possibly even each attempted update that uses a synchronization instruction) will lead to *cache invalidation* messages being sent to all other processors, these algorithms can overburden the caching system. This problem was identified by Mellor-Crummey and Scott in [MCS91], and a protocol, called the MCS-lock protocol, was proposed in which each acquire and release had a small constant number of accesses to remote memory locations. In this solution, a lock acquisition queue is constructed dynamically by swap instructions, and each processor spins on a different variable in a different cache frame. The first process in the queue holds the lock, and the next process in the queue will acquire the lock when the first process releases it. By assuring that a small constant number of cache-invalidation messages (one or two) will be sent by any acquisition or release of the lock, adding new processors does not significantly increase the bus and invalidation traffic caused by the locking mechanism—thus the locking mechanism is *scalable*.

One difficulty with the MCS-lock protocol is that locks are not recoverable. That is, it is possible to have a situation where the lock becomes unavailable when one or more of the processes holding or trying to acquire the lock die. In [BLS+95], we described a method for making simple spin locks *recoverable* without requiring kernel support, while maintaining the performance advantages of the spin lock over the kernel-supported lock. This ability to recover a spin lock is particularly useful for servers such as transaction processors which consist of several processes and which are intended to run indefinitely.

This paper is concerned with the recoverability problem for MCS-locks. We derive a protocol which allows the structure of these locks to be recovered on process failure. In this work, the integrity of the queue structure is the key difficulty in recovery, as opposed to the determination of ownership which was the central issue in our work on simple spin locks [BLS+95]. A key element in our recoverable spin lock algorithms in both this and our previous work is the use of a process to cleanup after dead processes. The *cleanup process*, which is assumed not to fail itself, releases locks the dead processes held and, in this work, repairs queue data structures.

In the *wait-free* data structure techniques, developed by Herlihy and others, (see, for example, [Her88, Her89]), compare-and-swap is used to make data structures resilient to process failure without the need for any form of cleanup. While we are designing a lock mechanism rather than a data structure, it is reasonable to ask if increased use of compare-and-swap can help. The full length version of this paper [BLS96] contains a modification of the MCS algorithm using compare-and-swap with a simple cleanup server. This algorithm is the only compare-and-swap-based adaptation of MCS we could find with a cleanup process simpler than the one described here. However, this algorithm does not guarantee a fixed number of remote references per lock acquisition and release, and thus lacks the scalability of the MCS-lock.

Thus, the use of a moderately more complex lock acquisition/release algorithm and cleanup server seems required to attain recoverability with scalability, even if compare-and-swap is available. Certainly, a solution such as the one presented in this paper is required on those architectures such as Sun's SPARC and HP's PA-RISC which do not have compare-and-swap instructions.

Recovering a mutual exclusion mechanism on process failure, the topic addressed by this paper, is

```
int compare-and-swap(register int *target,
                register int old, register int new)
    if (*target == old) *target = new; return 1;
    else return 0;

void swap(register int *target, register int val)
    int tmp=*target; *target=val; return tmp;
```

Figure 1: Definition of atomic instructions.

only part of an overall strategy of recovery from process death. Should a process die while waiting for or acquiring a lock, then recovery of the lock is sufficient to return the system to active use. However, if the process died while holding the lock, then the data structure covered by the lock must be returned to a consistent state before the lock is released. This may be accomplished by specialized routines (called Functional Recovery Routines, see [GR93]), which may well use database-style techniques for undo logging. If, for some data structures, such routines are prohibitively difficult, then at least the system may be brought down cleanly with a diagnosis of the problem. Note that the data structure covered by the lock could possibly be redesigned as a wait-free data structure, obviating the need for the recovery technique and even for the spin-lock itself. However, the purpose of this work is to add recoverability to algorithms and systems using standard critical sections without requiring such a monumental redesign.

The remainder of this paper is organized as follows. The problem is described in more detail in Section 2. In Section 3, we present the formal model that our algorithms are based on. In Section 4, we give an overview of our solution, followed by the new protocol in Section 5, including the cleanup procedure. We discuss correctness in Section 6. After presenting performance numbers in Section 7, we discuss related work and present our conclusions.

2 Problem statement

In Figure 1 we give the definition of the compare-and-swap and swap instructions by presenting them as functions. In an architecture supporting one of these instructions, each function must be implemented *atomically* by the corresponding instruction. Thus, all parameters to the function will be registers.

In [MCS91], two versions of the MCS-lock protocol are presented, one in which a compare-and-swap is used and one in which it is not. However, in both algorithms, swap is used to construct the queue, and compare-and-swap is only used for the transition during lock release from a queue with one entry to an empty queue. For clarity, we present the recoverability problem and its solution in terms of the simpler algorithm which uses compare-and-swap. (Our extension to handle the case where compare-and-swap is unavailable is included in [BLS96].)

The problem is to modify the protocol presented below so that the queue structure is maintained in case of multiple process failure. That is (informally),

- No queue entries are lost (i.e. no starvation due to a loss of queue structure).
- At most one process at a time believes it owns the lock.
- If an owner dies, ownership is transferred from the dead process to a live process in a "reasonable" amount of time.

We will give examples of how the failure of processes while executing the MCS-lock protocol can lead to the inability of other processes to acquire the lock.

In our adaptation of the MCS-lock algorithm, each process has a queue node, of type Qnode, to which mynode is a pointer. This queue node consists of a next pointer and a locked flag (which indicates whether this process has the lock). The value for locked is one of WAITING, OWNED or RELEASED,

```
acquire_lock(Lock *lock, register Qnode *mynode)
    Qnode *pred;
    mynode→next = NULL;
    mynode→locked = WAITING;
    pred = swap(&lock→tail,mynode);
    if (pred != NULL)
        pred→next = mynode;
        while (mynode→locked != OWNED);
    else
        mynode→locked = OWNED;
    return LOCK_ACQUIRED;


release_lock(Lock *lock, register Qnode *mynode)
    mynode→locked = RELEASED;
    if (mynode→next == NULL) //No one else in queue
        if (compare-and-swap(lock→tail,mynode,NULL))
            return LOCK_RELEASED; //No waiters
        while (mynode→next == NULL);
    mynode→next→locked = OWNED;
    //Give ownership to next process in the queue
    mynode→next = NULL;
    return LOCK_RELEASED;
```

Figure 2: MCS-lock protocol.

deviating slightly from the original protocol. A lock consists of a single field, tail, which points to the last node on the queue.

Figure 2 contains the lock acquisition and release routines as adapted from [MCS91]. The acquisition algorithm is straightforward. The swap with the lock pointer at the beginning of acquire simultaneously makes lock→tail point to mynode, while returning a pointer to the previous node to the calling process, P. P then makes the previous node's next pointer point to mynode. Once this second assignment takes place, a traversal of the list will once again end with the node pointed to by lock→tail (assuming all other such assignments have also taken place). At this point, P spins waiting for the previous owner of the lock to pass on ownership, or immediately assumes ownership if there was no previous owner.

Lock release code proceeds as follows. If P's entry in the list has a successor, then P just sets that node's flag to OWNED and unlinks its entry from the list by setting next to NULL. However, on releasing the lock, P's node (mynode) may not have a successor, in which case the algorithm must go to a little more trouble.

If mynode→next field is NULL, then either no other process is queued for the lock, or *perhaps* P's successor has performed the swap (and thus logically become the successor), but not yet patched P's next field. This is why a compare-and-swap is used to set lock→tail to NULL *only* if it is pointing to mynode. If the compare-and-swap succeeds, implying that lock→tail pointed to mynode, then P was alone in the queue and everything is fine. However, if the compare-and-swap fails (returns false), then some "interlopers" have come in and started adding themselves to the end of the queue. In this case, P just spins waiting for its own next field to be filled in by the first interloper.

This version of the algorithm is vulnerable to process failure in several places. For example, if a process dies while waiting for the lock, once it receives the lock it will never release it. Similarly while owning or releasing the lock, the death of the process will prevent ownership from being passed on. Particularly difficult is the window between the initial swap in the acquire code and the subsequent assignment to pred→next which links the new node into the queue. If a process has died after executing the swap, and before setting the next field, then the queue will become "broken" at that point. Thus, in pathological cases involving multiple process failures occurring in this same window, the queue becomes fragmented into separate lists.

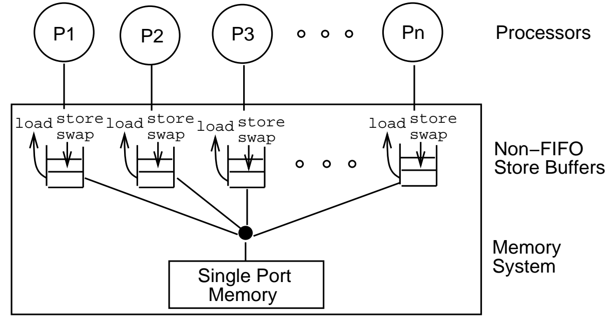

Figure 3: SPARC memory consistency model.

In attempting to recover this fragmented list, it is difficult to distinguish a `next` field which has not been set due to a dead process from one which has not been set due to a very slow process. The remainder of this paper addresses the problem of recovering the MCS-lock in case of process failure(s).

3 System model

We assume a hardware model with multiple processors sharing a common memory. To model the difficulties faced in implementing synchronization algorithms on modern processors, we assume a *weak memory model* in which writes issued by one processor can be seen in a different order by another processor, *unless* certain synchronization instructions are used. In particular, we take the *Partial Store Ordering* (PSO) model of [SFC91] (Figure 3) as the basis of our proofs of correctness. We augment the model with the concept of a `fence` instruction which forces all writes issued so far by the process issuing the `fence` to become globally visible. While the particular formalism we have chosen is intended to model the SPARC architecture, other popular processors implement similar consistency models, and we believe our solution would map easily to these other systems (though correctness might have to be re-proven).

We now present a brief overview of the Partial Store Order Model. This model is based on two partial orders, a partial ordering of memory operations generated by processor i (denoted by $\preceq^i$), and a partial ordering of memory operations executed at the global store (denoted by $\leq$). Informally, the axioms defining the effects of operations in this model are as follows.

Total order The store operations at the global store are totally ordered.

Atomic swap No other write to a global store memory location is allowed between the load and store parts of a `swap`.

Termination Writes buffered in the local store are eventually carried out at the global store.

Value The value returned by a load in processor i is the last store, in the $\leq$ order, of those stores that precede the load either in the $\leq$ order or in $\preceq^i$.

Load ordering $L_a^i \preceq^i Op_b^i \Rightarrow L_a^i \leq Op_b^i$ where Op is any memory operation, and L_a^i denotes a load from location a by processor i.

Storage Barrier Store operations from a processor that are separated by a 'storage barrier' or 'fence' instruction (`fence`) appear in the same order in $\leq$.

Same-location ordering Writes to the same location from the same processor are carried out in the order in which they were generated (formally, $S_a^i \preceq^i S_a'^i \Rightarrow S_a^i \leq S_a'^i$, where S_a^i denotes a store to location a from processor i).

We use the term `fence` to denote the generic storage barrier instruction. All shared memory systems of which we are aware with weak consistency guarantees (e.g. the Alpha [Cor92]) provide such `fence` instructions. In some cases, like the SPARC architecture, synchronization instructions must be used to

achieve the effect of fence, possibly at a higher cost. We use the term "integer" synonymously with the term "word", and assume that reads and writes of a word are atomic (also assumed in the PSO model). We assume that processes are fail-safe, i.e., they do not modify lock control information except through the provided interface code. We assume that an application can ask the OS whether a particular process has died, and that the cleanup process itself does not fail.

4 Overview

Our solution, which we term a Recoverable MCS-lock (RMCS-lock), adds four features to the original algorithm. This approach is similar in spirit to that taken in [BLS+95].

1. A pointer to a lock called wants is introduced to the Qnode structure. A process indicates its interest in a lock by setting its wants variable to point to the lock before attempting to acquire it, and not changing it until after it is released.

2. A boolean flag called volatile is also added to the Qnode structure. A process indicates that it is about to modify the structure of the queue by setting its volatile variable to true; it only resets this variable after the modification has finished, and all stores made by the modification have reached the global store.

3. A boolean flag (cleanup_in_progress) and an integer (clean_cnt) are added to the lock structure. Processes attempting to acquire or release the lock respect cleanup_in_progress, and take no steps which modify the queue when this flag is set, except for certain processes whose volatile flags are already set. The release routine examines clean_cnt to determine if the cleanup process has released the lock on behalf of a partially executed release.

4. Should a dead process's wants pointer be seen to point to a lock, the cleanup routine is invoked on that lock. This cleanup routine takes steps to ensure or restore the consistency of the lock queue.

We assume that processes make progress while their volatile flags are set. (We are not assuming that our concurrency mechanism is starvation free, just that once a process enters spin lock acquisition or release code, it receives some CPU time to execute, and if interrupted, it returns to the spin lock code within a finite amount of time.) A process may violate this assumption (e.g. by having a very low priority and getting no CPU time from the OS) in which case the technique of [BLS+95] can ensure progress by possibly killing some non-progressing processes.

Given the above assumption and the four additions to the algorithm, here is the outline of steps taken by the cleanup process upon finding that a process has died while accessing a lock. First, new processes are prevented from disturbing the lock's queue structure by setting cleanup_in_progress to true for the lock. Second, the routine waits for all processes which want the lock to set their volatile flag to false. At this point, the queue is stable, if inconsistent, and appropriate steps are taken to return it to consistency, including adding links where they are missing due to failed processes. Finally, cleanup_in_progress is set back to false, restoring the lock to normal operation. Note that the correctness of this procedure depends on the limited assumption of progress presented above, which guarantees that the cleanup routine will not wait forever for processes to set their volatile flags to false.

For simplicity of presentation, we assume that a process may hold only one spin lock at one time. This may be easily extended to allow multiple concurrent locks by the same process by preallocating several Qnode structures to each process. (Note that the allocation of the Qnodes cannot be brokered by our recoverable locks!) A process identifier stored in the Qnode, or some other mechanism must exist so that a Qnode can be mapped to a process by the cleanup server.

```
enum LockStatus {WAITING, OWNED, RELEASED};
struct Qnode {
    Qnode *next; LockStatus locked;
    Lock *wants; boolean volatile;
};
struct Lock {
    Qnode *tail; int clean_cnt;
    boolean clean_in_prog;
};
```

Figure 4: System data structures.

Note that while our technique still guarantees $O(1)$ remote references per lock acquisition or release during normal operation. However, *in case of process failure*, many other processes executing the protocol may spin on the single global cleanup_in_progress variable, causing a flurry of invalidation messages on the system bus when its value changes [MCS91]. However, each such episode of bus contention would correspond to an event of process failure and execution of the cleanup routine. These events are thousands of times more expensive than a single flurry of bus contention, and, unless a system is in dire circumstances, extremely rare. Thus, we chose not to introduce extra complexity into our protocols to retain the $O(1)$ remote reference property in the failure case.

5 Recoverable MCS-lock protocol

We now present the Recoverable MCS-Lock protocol. The data structures used for the scheme are summarized in Figure 4, and consist of a queue node (Qnode), which is a per-process entity, and a lock (Lock).

5.1 Recoverable MCS lock acquisition

The lock acquisition code is given in a C-like pseudo-code in Figure 5. The notable changes from Figure 2 involve the handling of the new volatile and wants fields of the Qnode structure. Note that both are set before the process adds itself to the end of the queue (A1-A2) *and* before checking the cleanup_in_progress flag. If cleanup_in_progress is seen to be true, then wants is set back to NULL and volatile to false (A3-A5), and the process waits until the cleanup is finished before returning failure to the caller (A6-A7).

If the process reaches A9, then the value false was seen for cleanup_in_progress. From this point, the algorithm is the same as in the original, except that volatile is set back to false *before* waiting for the locked flag to be changed to OWNED. This is consistent with the meaning of volatile, since a process waiting on the lock at A13 will not modify the queue structure again in that call to acquire_lock. Note that wants continues to point to this lock until the lock is released.

While the acquisition code would retry rather than returning an error if cleanup is detected, this complexity was omitted for ease of exposition.

5.2 Recoverable MCS lock release

The algorithm to release a lock, given in Figure 6, is more complicated than acquisition, mostly due to the code between R12 and R19. We will describe the case handled by this code after describing the simpler cases.

The first action of the release code is to set volatile to true, indicating an intention to adjust the queue structure. Next, the routine checks for cleanup_in_progress as in the acquire routine. Note that this code, in R1-R5 is symmetrical to the corresponding code in acquire, and that a similar piece of code occurs again in R16-R19.

```
acquire_lock(Lock *lock, register Qnode *mynode)
    mynode→next = NULL; Qnode *pred;
A1  mynode→volatile = true;
A2: mynode→wants = lock;
    ⟨fence⟩
A3: if (lock→clean_in_prog)
A4:     mynode→wants = NULL;
A5:     mynode→volatile = false;
A6:     while (lock→clean_in_prog) sleep(a little while);
A7:     return FAILED_TO_ACQUIRE_LOCK_TRY_AGAIN;
A8: mynode→locked = WAITING;
A9: pred = swap(lock→tail,mynode);
A10 if (pred != NULL)
A11:    pred→next = mynode;
        ⟨fence⟩
A12:    mynode→volatile = false;
A13:    while (mynode→locked != OWNED); // spin
    else
A14:    mynode→locked = OWNED;
        ⟨fence⟩
A15:    mynode→volatile = false;
A16 return LOCK_ACQUIRED;
```

Figure 5: Recoverable MCS lock acquisition.

```
release_lock(Lock *lock, Qnode *mynode)
R1: mynode→volatile = true;
    ⟨fence⟩
R2: if (lock→clean_in_prog)
R3:     mynode→volatile = false;
R4:     while (lock→clean_in_prog) sleep(a little while);
R5:     return LOCK_PARTIALLY_RELEASED_TRY_AGAIN;
R6: mynode→locked = RELEASED;
R7: if (mynode→next == NULL)
R8:     if (compare_and_swap(lock→tail,mynode,NULL))
R9:         mynode→volatile = false;
R10:        mynode→wants = NULL;
R11:        return LOCK_RELEASED;
R12:    int cc = lock→clean_cnt; //Cleanup proc sets flag
        ⟨fence⟩
R13:    mynode→volatile = false;
R14:    while (mynode→next==NULL&&cc==lock→clean_cnt
            && lock→clean_in_prog == false); // spin
R15:    mynode→volatile = true;
        ⟨fence⟩
R16:    if (lock→clean_in_prog||cc!=lock→clean_cnt)
R17:        mynode→volatile = false;
            mynode→wants = NULL;
R18:        while (lock→clean_in_prog) sleep(a little while);
            return LOCK_RELEASED;
R19 ⟨fence⟩
R20 mynode→next→locked = OWNED;
    //Give lock to next process in queue
R21 ⟨fence⟩
R22 mynode→volatile = false;
R23 mynode→wants = NULL;
R24 return LOCK_RELEASED;
```

Figure 6: Recoverable MCS lock release.

If mynode is seen to have a successor at R7, the routine proceeds with the simple case of passing ownership to the successor at R20. The fence instructions here are required for correctness, for example,

the one at R19 ensures that two processes never appear to be the owner at the same time to an outside observer.

If mynode has no successor at R7, then the algorithm attempts to change the lock tail to point to NULL after atomically ensuring that it still points to mynode at R8. Note that the need for this compare-and-swap instruction is relaxed by the MCS-lock algorithm. The corresponding relaxation for our algorithm can be found in [BLS96]. If the compare-and-swap succeeds, then the queue is empty and the algorithm notes that it is no longer modifying the queue structure by setting volatile to false (the compare-and-swap serves as a fence), and that it is no longer interested in the lock by setting wants to NULL.

We now describe the more complicated cases which may arise due to concurrent release and acquisition of the lock. If at R7 mynode→next was seen to point to NULL, but the lock tail is seen to be pointing to some other node than mynode at R8, then the compare-and-swap will fail. In this case, another process, P, has issued the swap at A9 of the acquisition algorithm, but not yet issued the assignment at A11. The process releasing the lock must then wait on P to set the releasing process's next field. The cc == lock→clean_cnt part of the check at R14 handles the case where P dies before executing A11, since otherwise R14 could possibly become an infinite loop. If the cleanup routine has started, it will take responsibility for freeing the lock from the releasing process (perhaps while the process spins at R18). The wait at R14 is done with the volatile flag unset (R13,R15) to ensure the key property that processes make progress while their volatile flags are set to true. (If a process P were to wait on another process while P's volatile flag is set, the second process could die, preventing P from progressing—violating this property.) Once volatile is set back to true, we must again check for cleanup_in_progress. This is necessary to maintain an invariant that processes cannot go from a non-volatile state to a statement which modifies the structure of the list while cleanup is in progress.

If a cleanup did take place between R13 and R18, then the cleanup routine has chosen the new owner for the queue. The clean_cnt, which would have been incremented by the cleanup routine, is used to make sure that the releasing process does not also attempt to pass on ownership, as this would corrupt the locking mechanism. The alternative strategy would have been to have the cleanup routine recognize processes in the middle of a release call, and try to work around them. This, however, would complicate our proof of correctness without simplifying the algorithm.

Once the node has a successor, the algorithm is back in the very first case, and the code continues with R19 as described above.

5.3 Cleaning up

Our mechanism assumes that a separate process or thread, known as the "cleanup process," is responsible for returning the lock to a consistent state in case of process failure. Further details concerning how this process initiates a cleanup and coordinates allocation and de-allocation of Qnode structures are presented in Section 5.4. We now describe the algorithm used to actually restore a lock to consistency.

The procedure cleanupMCSLock, given in Figure 7, proceeds as follows:

1. At C1, cleanup_in_progress for the lock is set to true, preventing processes which don't already have volatile set from modifying the list structure.

2. At C2-C3, a list of all parties interested in the lock is assembled into the set ViewWants, and the subset of those who may potentially be modifying the queue structure is placed into the set ViewVolatile.

3. Between C4 and C6, the cleanup process waits for ViewVolatile to empty out, so that no (live) process can be modifying the queue between C7 and C23.

4. Once all processes interested in the lock are no longer in volatile sections of the code, the clean_cnt for the lock is incremented, ensuring that a process which was releasing the lock will not continue to release it after the cleanup routine has assigned a new owner.

5. If the queue is empty at this point, the routine returns at C9 (freeing the Qnode for the process which died).

6. Between C10 and C16, the owner, if any, is determined. During this pass, all dead processes are also removed from ViewWants. If the owner is dead, then the structure guarded by the lock is suspect, and is cleaned up using the user-supplied routine cleanup_guarded_structures(). If the owner, or any waiting process, dies immediately *after* this check, it will be caught by a subsequent run of the cleanup routine.

7. At C17, the nodes for all live processes who are waiting on the lock, other than the one pointed to by the lock itself (end), are reassembled into a single list. This repairs any breaks in the list caused by dead processes, while safely ignoring the original order of the queue (thus the lock is not FIFO if a failure takes place).

8. At C18-C22, the owner, if any, is replaced at the head of the queue (else the node at the new head is made the owner), and the tail is placed at the end. The case of a single node on the list is handled at C22.

9. Finally, at C23, the lock is returned to active use by setting cleanup_in_progress back to false.

Note that, unlike [BLS+95], the process of determining the owner cannot be separated from cleaning up the guarded data structure. This is because the lock must be restored to sanity even if some process other than the owner died, and if the owner died, the lock must be cleaned up before the queue is returned to use.

5.4 Driver code and OS interaction

The cleanup process is started at system initialization and continues running while processing is taking place on the shared data. It periodically polls the OS for processes that have died. If a process has died with its wants field pointing to a lock, the cleanup process executes the procedure cleanupMCSLock (described in Section 5.3) on the in-doubt lock. This routine determines the owner of the lock, possibly calls a routine to recover the shared data if the lock was held by a dead process, and repairs any broken links that may exist because of process death. If the lock is in fact held by a dead process, cleanupMCSLock calls a user-supplied routine to return the shared data to a consistent state, and releases the lock. In all cases, the cleanup process releases the Qnode belonging to the dead process(es). Also, some processes in the system may find that they have been waiting much longer for a lock than they ought to have waited, and may notify the cleanup process which invokes cleanupMCSLock on the lock in question, rather than waiting for it to detect a possible process failure.

In order to ensure the correctness of the protocol in the face of multiple runs of the cleanup process, we must be somewhat careful. In particular, a process can die immediately after it has been included in the queue by the cleanup routine, but before the routine returns to the user. In this case, we must ensure that this dead process is caught by a *subsequent* run of the cleanup routine. A similar problem arises in the implementation concerning the allocation and deallocation of Qnode structures: if they are deallocated and subsequently reallocated during cleanup, it can easily cause trouble.

We handle both of these by insisting that the cleanup routine is in charge of deallocating Qnode structures. Further, we say that a process has been "seen to be dead" by the cleanup routine if IS_DEAD was called on that process at C12 and returned true. Any time the cleanup process notices a process is

```
cleanupMCSLock(Lock *lock)
 C1: lock→clean_in_prog = true;
     ⟨fence⟩
 C2: ViewWants = set of processes who want lock;
 C3: ViewVolatile = set of processes in ViewWants whose
                    volatile field is true;
 C4: while (ViewVolatile != ∅)
 C5:     foreach process P in ViewVolatile
             if (!P→volatile || IS_DEAD(P))
 C6:             remove P from ViewVolatile;
 C7: lock→clean_cnt++;
 C8: if (lock→tail == NULL)
         lock→clean_in_prog = false; ⟨fence⟩
 C9:     return ;
C10: owner = NO_PROCESS;
C11: foreach node Q in ViewWants
C12:     if (IS_DEAD(Q→pid))
             if (Q→locked == OWNED)
C13:             lock→cleanup_guarded_structures();
                 Q→locked = RELEASED;
C14:         remove Q from ViewWants;
         else if (Q→locked == OWNED)
C15:         owner = Q;
     end = NO_PROCESS;
     if (!IS_DEAD(lock→tail))
     //lock→tail is not NULL by previous check
C16:     end = lock→tail;
C17: (newHead,newTail) = reattach all Qnodes in ViewWants
                         with (Q→locked == WAITING),
                         except for "end", in arbitrary order;
     if (owner != NO_PROCESS)
C18:     owner→next = newHead;
     else
C19:     owner = newHead;
C20:     if (owner!=NO_PROCESS)
             owner→locked = OWNED;
     if (end != NO_PROCESS && end != owner)
C21:     newTail→next = end;
     else
C22:     lock→tail = newTail;
     ⟨fence⟩
C23: lock→clean_in_prog = false;
```

Figure 7: Cleanup code.

dead, it calls the cleanup routine, *unless* that process's wants pointer is null, or that process was *seen to be dead* by a previous run of the cleanup routine.

6 Correctness

The proof of correctness is omitted due to space constraints (see [BLS96] for proof). We do, however, provide some of the key intuitions from the proof. Unfortunately, even the exact statement of correctness is somewhat technical, but is summarized by the following conditions:

1. Ownership. Unless the queue is empty, there is a single owner at the end of cleanup, and it is the same process which owned the lock at the start, unless that process released the lock (or died). At no point is there more than one owner.

MCS-lock	1,011,000
Recoverable MCS-lock	354,000
RMCS-lock with no fence	606,000
SunOS 5.3 (Solaris) Semaphores	20,000
Recoverable Simple Spin Lock	526,000
Simple Spin Lock	2,128,000

Figure 8: Uncontested acquire and release per second.

2. Waiters. All waiters for the lock at the beginning (just before C2, but after the write of cleanup_in_progress has made it to all other processors) are either dead, waiting for the lock, or have acquired the lock at the end (just before C23). No other processes are added to the lock queue.

3. Dead processes. All processes seen to be dead by the cleanup routine are removed from the queue.

We now present a few key points of intuition concerning the proof of correctness.

1. ViewWants holds all candidates for owning or waiting on the lock; ViewVolatile (a subset of ViewWants) holds all candidates for modifying the lock. This follows from the careful manner in which the wants and volatile flags are set, as well as the cleanup_in_progress flag. Note that these sets are overestimates, and an individual process may no longer want the lock nor be able to modify it.

2. Thus, when ViewVolatile becomes empty at C7, no live process may be modifying the queue, and ownership may not change.

3. The remainder of the cleanup algorithm, when working on this stable though possibly inconsistent structure, correctly repairs the queue, and if necessary chooses a new owner.

4. Any process seen as dead by the cleanup routine is not included in the queue.

7 Performance

We have implemented both the original MCS locking algorithm [MCS91] and our recoverable MCS algorithm for SPARC processors. We tested the speed of the algorithms under no contention by running them on a 60MHZ SPARCStation 20/61. For a fence on the SPARC, we used swap to a dummy memory location. This has a significant impact on performance, which we will discuss below. The performance results, as well as results from [BLS+95] for comparison, are presented in Figure 8. (Our previous recoverable spin lock algorithm [BLS+95] is labeled "Recoverable Simple Spin Lock.") The numbers represent uncontested acquire and release pairs per second. The tests indicate that adding recoverability to the MCS-lock algorithm on the SPARC decreases performance by a factor of three. This is slightly better than the factor of four hit for regular spin locks due to the complexity of the original MCS-lock algorithm. The resulting figure of 354,000 pairs per second is still over an order of magnitude better than OS semaphores.

The number titled "with no fence" represents the speed with fence instructions implemented as nop rather than a swap. This is not a correct protocol, but it does indicate that a significant portion of the performance hit of the algorithm comes from the cost of the swap instructions. Thus, the recoverable protocol may be significantly less costly on a machine with a relatively inexpensive fence instruction.

From previously published MCS timing results [MCS91], we expect that our Recoverable MCS algorithm will be far more scalable in the high contention case than our recoverable spin lock [BLS+95].

It is only a factor of three slower than the original MCS algorithm—a low overhead if both scalability and recoverability are needed, since the alternatives are to shut down the system on process death, to use system semaphores, or to use recoverable test-and-set spin locks [BLS+95], which would be far less scalable.

8 Related work

Our previous work [BLS+95] produced high-performance, recoverable spin locks. Under low contention, our spin locks are roughly 25 times faster than recoverable system semaphores, but about four times slower than non-recoverable spin locks. This work extends that previous work by improving cache performance under high contention. The new algorithm is based on the MCS lock algorithm of Mellor-Crummey and Scott [MCS91], and can be seen as an effort to make that lock recoverable. Other previous work on improving the cache performance of spin locks appears in [Cra93, GT90]. However, none of these previous algorithms were recoverable. In [MR95], the authors present techniques for making locks recoverable. However, they require hardware instructions to lock/release a cache line, but most commercial systems do not provide these instructions. Requiring only an atomic swap instruction, our protocol can be implemented on a large collection of architectures.

A large body of work by Herlihy and others exists on *wait-free* data structures (see, for example, [Her88, Her89]). In these techniques, also motivated by similar concerns about process failure and slow-down, a process attempts to take a data structure from one consistent state to another with a single compare-and-swap instruction. Turek, Shasha and Prakesh [TSP92] show a general technique for making a class of algorithms resilient to process failure. Both techniques require rewriting all concurrent data structures using a different paradigm, and require architectures with compare-and-swap instructions. Our work is aimed at providing similar failure-tolerance properties to more standard mutual exclusion-based concurrency control mechanisms. Also, several of our techniques work on machines with less powerful hardware abstractions than compare-and-swap.

Older, software-based approaches to mutual exclusion that do not rely on atomic instructions are trivially recoverable. For example, the standard starvation free software mutual exclusion algorithm, [EM72], uses an array of per-process state information, thus the state of any process with respect to a given spin lock can be determined immediately by inspecting that variable. Any interference with other processes can be removed by resetting the state to "uninterested." However, these algorithms are very expensive in time and space, and would apparently scale poorly. All these algorithms require time at least proportional to the number of processes to acquire an *uncontested* spin lock, while solutions based on synchronization hardware, such as ours, typically require a small constant number of accesses. Furthermore, these systems require space proportional to the number of processes times the number of spin locks, as opposed to the sum of the two, as with our algorithm.

9 Conclusions and future work

We have extended our previous work on recoverable, high-performance spin locks for shared memory multi-processors [BLS+95] by producing an algorithm (based on the MCS-lock protocol [MCS91]) that has good cache performance as the number of processors increases. Our current algorithm has comparable performance to our previous algorithm in the low contention case. The recoverable algorithm represents a good solution for mutual exclusion on a machine with a large number of processors when fault tolerance and recovery from process failure is desired.

From these two pieces of work, it appears that using a cleanup process to make spin locks recoverable is a generally useful technique, as is writing globally accessible control information before and after spin lock acquisition. In our future work, we hope to derive general principles from these two pieces of work

that can be used to make any locking algorithm on which certain properties hold recoverable. We would also like to compare the performance of a variety of recoverable and non-recoverable algorithms on a shared memory multi-processor with a large number of processors under both high and low contention.

10 Acknowledgments

We would like to thank S. Sudarshan for several useful discussions on proof techniques for this protocol (any errors are ours).

References

[And90] 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.

[BLS+95] P. Bohannon, D. Lieuwen, A. Silbershatz, S. Sudarshan, and J. Gava. Recoverable user-level mutual exclusion. In *Proc. IEEE Symp. on Parallel and Dist. Processing*, October 1995.

[BLS96] P. Bohannon, D. Lieuwen, and A. Silberschatz. Recovering scalable spin locks. Technical Report 112580-960626-04, Bell Laboratories, June 1996.

[Cor92] Digital Equipment Corporation. *The Alpha Architecture Handbook*, 1992.

[Cra93] Travis S. Craig. Queuing spin lock algorithms to support timing predictability. In *Proc. Real-Time Systems Symp.*, pages 148–157, December 1993.

[EM72] M. A. Eisenberg and M. R. McGuire. Further comments on Dijkstra's concurrent programming control problem. *Communications of the ACM*, 15(11), November 1972.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, CA, 1993.

[GT90] Gary Graunke and Shreekant Thakkar. Synchronization algorithms for shared memory multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990.

[Her88] Maurice Herlihy. Impossibility and universality results for wait-free synchronization. Technical report, CMU, TR-CS-88-140, May 1988.

[Her89] Maruice Herlihy. A methodology for implementing highly concurrent data structures. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, March 1989.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[MR95] L. Molesky and K. Ramamritham. Recovery Protocols for Shared Memory Database Systems. *Proc. SIGMOD*, pages 11–22, May 1995.

[SFC91] Pradeep Sindhu, Jean-Marc Frailong, and Michel Cekleov. Formal specification of memory models. Technical Report CSL-91-11 [P91-00112], Xerox Corporation, December 1991.

[TSP92] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. In *Proc. Principles of Database Sys.*, June 1992.