A Simple Local-Spin Group Mutual Exclusion Algorithm*

Patrick Keane and Mark Moir
Department of Computer Science
University of Pittsburgh

Abstract

This paper presents a new solution to the *group mutual exclusion* problem, recently posed by Joung. In this problem, processes repeatedly request access to various "sessions". It is required that distinct processes are not in different sessions concurrently, that multiple processes may be in the *same* session concurrently, and that each process that tries to enter a session is eventually able to do so. This problem is a generalization of the mutual exclusion and readers-writers problems.

Our algorithm and its correctness proof are substantially simpler than Joung's. This simplicity is achieved in part by building upon known solutions to the more specific mutual exclusion problem. Our algorithm also has various advantages over Joung's, depending on the choice of mutual exclusion algorithm used. These advantages include admitting a process to its session in constant time in the absence of contention, spinning locally in Cache Coherent (CC) and Non-Uniform Memory Access (NUMA) systems, and improvements in the complexity measures proposed by Joung.

1 Introduction

In the *group mutual exclusion* problem recently posed by Joung [12], N processes repeatedly attend "sessions" labeled from $\{1 \ldots M\}$. Processes that have requested to attend the same session may do so concurrently. However, if two processes request different sessions, then they may *not* attend their sessions at the same time. It is required that any process that requests to attend a session is eventually able to do so. It is further required that if a process p has requested a session t, and no process is currently attending or requesting a different session, then p can attend session t without waiting for other processes to leave the session. This requirement precludes the direct application of an ordinary mutual exclusion algorithm to solve the problem, because it requires that processes can attend their sessions concurrently.

The main difficulty associated with designing group mutual exclusion algorithms arises from the need to balance the three correctness requirements. In particular, the requirement that processes requesting the same session can attend that session concurrently conflicts with the requirement that a process that requests a session eventually attends it. This is because if a process p requests a session t while other processes are attending session t, then p should be able to attend t too. However, always admitting a process that requests the current session could lead to starvation of processes that are waiting for a different session. We therefore need a mechanism for "closing" a session so that even if new processes request to attend the current session, they will be forced to wait, thereby allowing different sessions to be established.

Group mutual exclusion is a natural generalization of some well-known and well-studied synchronization problems. In particular, the group mutual exclusion problem is a generalization both of the mutual exclusion problem [5, 7, 10, 14, 16, 18] and of the readers/writers problem [6]. To see this, observe that, given a group mutual exclusion algorithm, we can solve the mutual exclusion problem by having each process use its process id as a session number. We can solve the readers/writers problem by having each writer request a different session, and having all read-

*Email: {patrick,moir}@cs.pitt.edu. Work supported in part by an NSF CAREER Award, CCR 9702767.

ers request the same special session. This allows readers to attend the session concurrently, while ensuring that each writer attends in isolation. Group mutual exclusion also has some similarity to k-exclusion [2, 4, 8, 9] in that it allows multiple processes to execute in a session concurrently (we can think of attending a session as executing in a critical section). However, k-exclusion imposes a bound on the number of processes that may attend concurrently, and does not distinguish between these processes.

An efficient solution to the group mutual exclusion problem is useful in applications where a single resource is shared amongst a number of processes, and can be used concurrently for some purposes, but not for others. An example, given by Joung [12], is a system with a shared CD jukebox. By assigning a session number to each CD, a group mutual exclusion algorithm would allow multiple processes to read the currently-loaded CD concurrently, while forcing processes that require a different CD to wait. If a purely mutually-exclusive approach were used, the system could be forced to repeatedly load and unload two CDs in the face of alternating requests. In contrast, a group mutual exclusion algorithm "groups" together concurrent requests for the same CD, thereby reducing the frequency of CD switches, and improving utilization of the CD jukebox.

Joung presented a solution to the group mutual exclusion problem [12] that is based on only read and write operations. Joung's solution is reminiscent of early mutual exclusion algorithms and shares several of their disadvantages. In particular, the solution is based on an array of flags. In order to reach the critical section (that is, to attend a session), a process must read the flags of all other processes, often more than once. As a result, this solution has high time complexity, even in the absence of contention. As pointed out by Lamport [15], this is a common case, so it should be as efficient as possible.

A further disadvantage of Joung's algorithm arises from the mechanism employed to effect waiting. In particular, while a process is waiting to attend a session, it must repeatedly read the flags of all other processes, and in some cases, repeatedly write shared variables. A number of recent performance studies [4, 5, 10, 16, 18] have shown that minimizing traffic on the processor-to-memory interconnect is crucial for scalable performance of synchronization algorithms for shared-memory multiprocessors. The reason is that waiting mechanisms such as the one used in Joung's algorithm generate excessive amounts of traffic on the processor-to-memory interconnect, which quickly becomes a bottleneck, thereby degrading overall system performance. It is clear that these issues will be equally relevant to the design of efficient and scalable group mutual exclusion algorithms.

Previous work in addressing the problem of excessive interconnect traffic has focused on two classes of shared-memory multiprocessor systems, namely cache coherent (CC) and non-uniform memory access (NUMA) systems [4, 16, 18].[1] This work has distinguished between *remote* and *local* memory references; *remote* memory references are references that require a traversal of the processor-to-memory interconnect. In a *local spin* algorithm, all waiting mechanisms generate a bounded number of remote memory references, regardless of how long they wait. As discussed in more detail later, complexity measures for counting the number of remote references have been developed for both CC and NUMA systems, and performance studies have verified that such complexity measures accurately reflect and predict the scalability of mutual exclusion algorithms [4, 16, 18].

In this paper, we present an efficient group mutual exclusion algorithm that is designed to spin locally in both CC and NUMA systems. Moreover, depending on the configuration of our algorithm, it can also ensure that processes are able to attend their sessions in constant time in the absence of contention. In contrast, Joung's algorithm generates an unbounded number of remote references in both classes of systems, and forces a process to do $O(N)$ work before attending its session, even in the absence of contention.

Our algorithm is much simpler than Joung's and our proofs are both simpler and more formal. This is largely due to the fact that we based our algorithm on known solutions to the less general mutual exclusion problem. Our algorithm can be instantiated with different mutual exclusion algorithms (possibly including algorithms that have not yet been discovered) to yield group mutual exclusion algorithms with different properties. In the paper, we discuss several known mutual exclusion algorithms and the properties our algorithm has when instantiated with these algorithms. From a theoretical standpoint, none of the resulting algorithms are *strictly* superior

[1] IN [4], NUMA systems were called distributed shared memory (DSM) systems.

to Joung's with respect to the complexity measures he proposes to use for evaluating group mutual exclusion algorithms. In fact, our use of mutual exclusion to solve the group mutual exclusion problem means that our solution can *never* yield a strictly superior algorithm. Nonetheless, our approach does yield several practical advantages over Joung's algorithm, and as discussed below, comes very close to providing algorithms that are theoretically superior too.

Table 1 compares Joung's algorithm, our algorithm instantiated with various mutual exclusion algorithms, and a third approach that depends on wait-free universal constructions; the latter approach is discussed in more detail in Section 5. (Our algorithm can be instantiated with any starvation-free mutual exclusion algorithm to yield a correct group mutual exclusion algorithm; we have chosen to discuss the algorithms shown in Table 1 because each yields a different advantage over Joung's algorithm.)

The "Primitives" column shows the shared-memory primitive operations used. The "CC" and "NUMA" columns show for each class of systems the worst-case number of remote references generated by each algorithm in attending and leaving a session once. Where no worst-case bound exists, the table shows infinity. The next column shows the worst-case time complexity of attending and leaving a session once in the absence of contention. The last three columns refer to complexity measures proposed by Joung [12]. Roughly speaking, context switch complexity measures the worst-case number of different sessions that can be established while a process is waiting to enter a session; time complexity measures the worst-case number of passages a process must wait before attending its session (a passage is the interval during which a process attends a session); and concurrency measures the maximum number of processes that can join the current session before a particular process leaves the session if some process is waiting for a different session. These measures are discussed in more detail in Section 4.

As the table shows, all of the algorithms shown have some advantage over Joung's algorithm. In particular, all of them spin locally in at least CC systems, all have contention-free time complexity that is the same as or better than Joung's, and most have a lower concurrency measure. As Joung points out, higher concurrency may imply better resource utilization because more processes can join a session before a new session must be started. However, as discussed in Section 4, our algorithm can easily be modified to provide a higher concurrency measure, whereas it is not easy to modify Joung's algorithm to reduce it. Our mutual-exclusion-based solutions have at best $O(N)$ context switch complexity (and this is inherent to our approach), while Joung's algorithm has $O(M)$ (recall that M is the number of possible sessions). Thus, if $N >> M$, Joung's algorithm has an advantage over ours in terms of this measure. For this reason, our algorithm is not strictly superior to Joung's from a theoretical standpoint. Nonetheless, we believe that our approach will be preferable for most applications. Finally, depending on the choice of mutual exclusion algorithm, our approach can achieve $O(N)$ time complexity (as defined by Joung), while Joung's has $O(NM)$, making our approach strictly better than Joung's in this respect. In Section 5 we discuss an approach based on wait-free constructions that yields a group mutual exclusion algorithm that is superior to Joung's with respect to all measures discussed here (except that it is based on strong synchronization primitives).

The remainder of this paper is organized as follows. In Section 2, we formally define the group mutual exclusion problem and present definitions that are used in the rest of the paper. In Sections 3 and 4, respectively, we present our group mutual exclusion algorithm and discuss its analysis with respect to the complexity measures discussed above. Then, in Section 5 we discuss how wait-free constructions can be used to avoid unnecessary waiting imposed by the use of mutual exclusion in our algorithm. Concluding remarks appear in Section 6. A correctness proof for our algorithm is presented in [13].

2 Preliminaries

In this section we define the group mutual exclusion problem more precisely and describe the complexity measures we use to evaluate solutions to this problem. Joung's measures are discussed in Section 4.

We assume N processes, each of which cycles through a non-critical section, an entry section, a critical section, and an exit section. Each time a process p transits from its non-critical section to its entry section, p non-deterministically chooses a "session" $p.t$ from $\{1 \ldots M\}$. Every process that is executing outside its critical section eventually takes another step.

Algorithm	Primitives	Remote References CC	NUMA	Cont. Free	Context Switch Complexity	Time Complexity	Concur.
Joung [12]	R/W	∞	∞	$O(N)$	$O(M)$	$O(NM)$	$O(N^3)$
Lamport [14]	R/W	$O(N)$	∞	$O(N)$	$O(N)$	$O(N)$	$O(N)$
Y&A (1) [18]	R/W	$O(\log N)$	$O(\log N)$	$O(\log N)$	∞	∞	∞
Y&A (2) [18]	R/W	$O(N)$	$O(N)$	$O(1)$	∞	∞	∞
MCS [16]	CAS/FAS	$O(N)$	$O(N)$	$O(1)$	$O(N)$	$O(N)$	$O(N)$
Trans. [17]	LL/SC	$O(N)$	∞	$O(1)$	$O(\min(N, M))$	$O(N)$	$O(N)$

Table 1: The first row shows Joung's algorithm, the next four rows show our algorithm instantiated with various mutual exclusion algorithms, and the last row shows a third approach using wait-free transactions (see Section 5).

The group mutual exclusion problem is to design code for the entry and exit sections in order to ensure that the following properties hold in all executions.

Mutual Exclusion: If processes p and $q \neq p$ are executing in their critical sections, then $p.t = q.t$.

Concurrent Entering: If process p requests a session $p.t$ and no process q such that $q.t \neq p.t$ is executing outside its non-critical section, then p eventually reaches its critical section (even if no other process leaves its critical section).

Bounded Waiting: Any process that leaves its non-critical section eventually reaches its critical section, and any process that leaves its critical section eventually returns to its non-critical section.

A *local spin* algorithm generates a bounded number of remote memory references in the worst case. We use the definitions for remote references proposed in [4]. These definitions consider two classes of systems: cache-coherent (CC) systems and non-uniform memory access (NUMA). Briefly, a memory reference is *local* in a NUMA system if the memory word referenced is physically located in the processor's local partition of shared memory, and *remote* otherwise. In a CC system, the definition is somewhat more complicated. Essentially, we count a loop in which we read a value until it changes as generating 2 remote references: one for the first read to establish a cached copy, and one for the read after the value changes because this cache copy has been invalidated. This model is explained in more detail in [4]. For each group mutual exclusion algorithm, we measure the worst-case number of remote references generated by a process executing the entry and exit sections once.

3 Algorithm

Our group mutual exclusion algorithm is shown in Figure 1. We assume that there is a statement 0 (not shown), which non-deterministically selects a session t and transfers control to statement 1. In addition, after executing statement 22, a process returns to statement 0. The shared variables *Num* and *Session* refer to the current session, or the most recent session if there is none. Roughly speaking, *Num* counts the number of processes within the critical section, and *Session* records the current session identifier. The simplicity of our algorithm is due in part to the use of a single shared lock M, which restricts access to other shared variables. The FIFO queue Q records which processes are waiting for a new session. The two remaining variables — *Need* and *Wait* — are arrays of size n. *Need*[p] stores the session being requested by process p, and *Wait*[p] is a boolean variable on which process p spins if it cannot immediately enter its critical section.

To request a session, a process p first initializes its spin location to false (line 1), and then "announces" the session it is requesting (line 2). Then, after acquiring the lock M (line 3), p checks whether the current session is the session that it is requesting, and if so, whether the session is still "open" (i.e. there are no processes waiting in the queue for a different session). If this is the case, then p increments the counter, *Num*, to indicate that another process has joined the critical section (line 5), and proceeds directly to the critical section. Otherwise, p checks to see if the current session has no processes in it (line 6). In this case, p establishes the session that it is requesting (line 7), records that there is one process (namely p) in the session (line 8), and proceeds to

```
shared variables
    M: lock;  Session, Num: integer;  Q: queue of 0..N-1;
    Wait: array [0..N-1] of boolean;  Need: array [0..N-1] of integer
local variables
    t, v: integer;
initially
    Num = 0 ∧ Session = 1 ∧ Q = ∅

1:  Wait[p] := false;
2:  Need[p] := t;
3:  Acquire(M);
4:  if Session = t ∧ Q = ∅ then
5:      Num := Num+1
6:  else if Session ≠ t ∧ Num = 0 then
7:      Session := t;
8:      Num := 1
    else
9:      Wait[p] := true;
10:     Enqueue(Q, p)
    fi;
11: Release(M);
12: while Wait[p] do od;

<<Attend session t>>

13: Acquire(M);
14: Num := Num-1;
15: if Q ≠ ∅ ∧ Num = 0 then
16:     Session:= Need[Head(Q)];
17:     for each v ∈ Q do
18:         if Need[v] = Session then
19:             Delete(Q, v);
20:             Num := Num+1;
21:             Wait[v] := false
    fi od fi;
22: Release(M)
```

Figure 1: A simple local-spin group mutual exclusion algorithm. Code is shown for process p.

the critical section. If neither of the tests at lines 4 or 6 succeeds, then either the current session is not "open" or a session other than the one p is requesting is in progress (i.e., some processes are currently in that session). In the first case, p cannot enter the critical section, as the current session is no longer accepting additional processes (allowing new sessions to be established). This is necessary to ensure bounded waiting. In the second case, p cannot enter the critical section because doing so would violate mutual exclusion. Therefore, in both cases, p must wait: it first sets its spin location (line 9), and then enqueues itself into Q (line 10). As explained below, this ensures that some process will eventually establish the session for which p is waiting, and then clear p's spin location, thereby allowing p to enter the critical section.

When a process p *leaves* the critical section, after acquiring the lock M (line 13), it decrements Num to indicate that is has left (line 14). Next, p checks to see if there are processes waiting for a new session and whether it is the last process to exit the critical section (line 15). If this is the case, then p starts a new session and releases all processes waiting for that session, so that they can enter (lines 16 to 21). To achieve this, p examines the head of Q to find the first process that is waiting for a new session, determines which session that process is waiting for, and establishes that session (line 16). Next, p traverses Q, checking each process in Q to see whether it is requesting the newly-established session (line 18). If so, p removes that process from Q (line 19), increments Num to indicate that the process has been added to the session (line 20), and then releases the process from waiting so it can enter the critical section (line 21). Thus, all processes that are in Q waiting for the newly-established session are admitted to that session.

Joung's algorithm has a similar mechanism — which he calls "capturing" – for admitting multiple waiting processes to a newly-established session. With this mechanism, the *first* process to enter a session checks whether there are other processes also requesting the same session and captures any such process into that session. In Joung's algorithm, however, capturing is achieved by checking a flag for *every* other process, even if it has no interest in entering a session. This explains why Joung's algorithm has $O(N)$ time complexity, even in the absence of contention. In contrast, our algorithm checks only those processes that are in Q.

Below we give a brief and informal overview of the proof that our algorithm satisfies the mutual exclusion, concurrent entering, and bounded waiting properties. Full proofs are provided in [13].

First, only processes that have requested the same session can execute within their critical sections concurrently (i.e., our algorithm satisfies the mutual exclusion property), because a process p that is requesting a session t can only enter or execute within the critical section if $Session = t$. To see this, note that if p enters the critical section because one of the tests at lines 4 and 6 succeeds, then either $Session = t$ already holds (line 4), or p establishes $Session = t$ (line 7). Also, if p enters the critical section through lines 9..10, then some process q must have cleared $Wait[p]$. In this case, because q only releases processes that are waiting for the session it has just established (see lines 16 and 18), $Session = t$ when p enters the critical section. Also note that *Session* changes only when $Num = 0$ (see lines 7 and 16), which means that no process is in the critical section or about to enter it.

Second, we argue that processes requesting the same session can attend their sessions concurrently provided there are no other processes waiting for a different session (i.e., our algorithm satisfies the concurrent entering property). Because our algorithm uses only starvation-free mutual exclusion algorithms to implement the *Acquire* and *Release* procedures, and because every process outside its noncritical section eventually takes another step, p eventually acquires the lock at line 3, even if no process leaves the critical section (of the group mutual exclusion algorithm). Also, p waits at line 12 only if the tests at lines 4 and 6 both fail — that is, either $Session = t \wedge Q \neq \emptyset$ holds or $Session \neq t \wedge Num > 0$ holds. In the first case, some process is waiting in Q. Because $Session = t$, some process must be waiting for a different session (so p waiting does not violate concurrent entering). In the second case, at least one process is in the critical section for a session other than t. Finally, a process that requests a session eventually attends the session (i.e., our algorithm satisfies the bounded waiting property). To see this, recall that a process is prevented from entering the critical section directly only if either $Session = t \ \wedge \ Q \neq \emptyset$ holds or $Session \neq t \wedge Num > 0$ holds. In both cases, it can be shown that some process is in or about to enter the critical section. Also, because p puts itself in Q in this case (line 10), no more processes can join the current session. Therefore, when the last process in the critical section leaves, it will decrement *Num* to zero, remove a process from Q, and establish its session by releasing *all* processes waiting for that session into the critical section (lines 17 to 21). Repeating this argument, p will eventually reach the head of Q and be released into the critical section (unless it is first captured into its session because another process ahead of p in Q requested the same session).

4 Analysis

In this section, we discuss the analysis our of algorithm with respect to the time complexity measures described previously, and briefly explain how the bounds shown in Table 1 are calculated.

The analysis of our algorithm depends on the choice of mutual exclusion algorithm used to implement the *Acquire* and *Release* procedures. Before addressing this point, we first discuss the components of the analysis that are independent of this choice.

We first consider the entry section. The number of remote references generated by statements 1..2 and 4..10 is constant. (Note that it is easy to implement constant-time operations on Q because all such operations are invoked within the *Acquire* and *Release* of M, so Q can be implemented by sequential queue code.) We assume that $Wait[p]$ is stored in a partition of shared memory that can be accessed locally by p in a NUMA system, so the loop at statement 12 does not generate any remote references in such systems.[2] Also, by the assumptions about CC systems stated in Section 2, this loop generates at most 2 remote references in a CC system. (Note that only statement $p.9$ sets $Wait[p]$, so if $Wait[p]$ becomes false while p is executing the loop at statement 12, it remains false until p exits the loop.)

Next, we consider the exit section. The number of remote references generated by statements 14 through 16 is constant. Statements 17 through 21

[2] In practice, it is often cumbersome to implement an array that has its elements distributed across different partitions of shared memory. However, our algorithm treats the *Wait* variable as an array only to simplify exposition of the algorithm and its proof. In fact there is no need for these locations to be stored in consecutive memory locations provided each process can efficiently find the spin location of each other process. This could be achieved with one remote reference by using a static array of pointers, or even with no remote references at all by making a redundant copy of such a table in each partition in advance.

constitute a single traversal of Q, deleting selected elements. Each element can be examined and deleted if necessary in constant time. Thus, because the maximum length of Q is N (actually $N-1$ as the process executing the loop is not in Q), this loop generates $O(N)$ remote references in the worst-case in both CC and NUMA systems. Thus, apart from the remote references generated by the *Acquire* and *Release* procedures, the total number of remote references generated by one critical section acquisition and release is $O(N)$ in the worst case in both CC and NUMA systems.

In the absence of contention, $Num = 0 \ \wedge \ Q = \emptyset$ holds when a process evaluates the tests at statements 4 and 6. One of these tests must succeed, so lines 1..2 and 4..10 have constant time complexity in the absence of contention. Furthermore, in the absence of contention, Q is empty when the exit section is executed, so lines 14 to 21 also have constant time complexity.

We now consider the *Acquire* and *Release* procedures. If we use Mellor-Crummey and Scott's algorithm (MCS) [16], these procedures produce a constant number of remote memory references in the worst case in both CC and NUMA systems. MCS also has constant time complexity in the absence of contention. However, the MCS algorithm depends on the availability of strong synchronization primitives. In the absence of such primitives, we can instead use mutual exclusion algorithms that are designed to use only read and write operations. There are numerous such algorithms. Here, we consider algorithms of Lamport [14] and of Yang and Anderson [18] because each of these algorithms, when combined with our algorithm, yields a group mutual exclusion algorithm with some advantage over Joung's.

First, the waiting mechanism in Lamport's algorithm does not write any shared variables. Furthermore, this mechanism reads only $O(N)$ shared variables, and stops reading each one after it changes a constant number of times. Therefore, using this algorithm together with our algorithm yields a group mutual exclusion algorithm that spins locally in cache-coherent systems (in fact, it performs at most $O(N)$ remote references per critical section acquisition). However, Lamport's algorithm performs an unbounded number of remote references in a NUMA system. Furthermore, in Lamport's algorithm, a process takes $O(N)$ time to reach the critical section even in the absence of contention.

Yang and Anderson presented two mutual exclusion algorithms, both designed to spin locally on either a CC or a NUMA system. Their first algorithm generates $O(\log N)$ remote references in the worst case and also has $O(\log N)$ time complexity in the absence of contention. Their second algorithm has $O(1)$ time complexity in the absence of contention, but generates $O(N)$ remote references in the worst case. This concludes our discussion of remote reference and contention-free time complexity. For a summary, see Table 1.

We now consider the complexity measures proposed by Joung [12]. He defines these measures somewhat informally, using phrases such as "from the time a process completes its request for a [session] until the time it initiates a passage through the [session]". It is not clear what this and other phrases used by Joung mean in the context of an arbitrary group mutual exclusion algorithm. We therefore attempt to compare our algorithm to Joung's with respect to his complexity measures using the most reasonable interpretation we can think of.

First, we consider "context switch complexity". For this measure, Joung considers the interval between the time a process p "completes its request" for a session until the time p "initiates a passage through" the session, and measures context switch complexity as the number of sessions that can be established in this interval. We formalize this interval as beginning at the first time p is "blocked" (that is, if run alone from this state, p will return infinitely often to this state without ever entering its session) until the first time p is "enabled" (that is, regardless of the actions of other processes, p can attend a session in a finite number of its own steps [8, 9]). Because our group mutual exclusion solution depends on the use of a mutual exclusion algorithm, the first point in this interval may be in the entry section of the mutual exclusion algorithm. Because the mutual exclusion algorithms of Lamport and of Mellor-Crummey and Scott have the first-come-first-served property,[3] it follows that at most $N-1$ processes can enter a session before p acquires the lock. Because p has not

[3] More accurately, Lamport's algorithm has the first-in-first-enabled property [2]. This property ensures that a process p can execute an initial "doorway" without waiting, and that p will reach the critical section (of the mutual exclusion algorithm) before any process q that starts executing the doorway *after* p completes executing the doorway.

yet acquired the lock and added itself to Q, these $N-1$ processes could each establish a new session. After p has acquired the lock, p either joins a session or adds itself to Q. From this point, at most $\min(N-1, M-1)$ new sessions can be established, before p either reaches the front of the queue and has its session established or is "captured" into a session established by another process. Thus, using either Lamport's or Mellor-Crummey and Scott's algorithm, our algorithm yields context switch complexity of $O(N + \min(N-1, M-1)) = O(N)$. On the other hand, the algorithms of Yang and Anderson do not have the first-in-first-enabled property. They therefore do not ensure bounded context switch complexity, because a process p may be "overtaken" an unbounded number of times in the *Acquire* procedure, and therefore an unbounded number of sessions can be established before p enters its session.

We next consider Joung's "time complexity" measure. This measure considers the same interval as the context switch complexity measure, and counts the size of the minimal cover of all "passages" that can occur during this interval. Unfortunately, a "passage" is defined by Joung as "the interval during which a [process] passes through the [critical section]". This does not make clear whether the interval includes executing the exit section, or merely leaving the critical section. If we make the former assumption, our analysis is easier and our bounds lower. We therefore make the latter assumption to be on the conservative side. With these definitions in mind, we can discuss the analysis of our algorithm with respect to Joung's "time complexity" measure. For the same reasons as before, using Yang and Anderson's algorithm does not provide a bound on this measure. Again, using either Lamport's or Mellor-Crummey and Scott's algorithm, $O(N)$ processes can attend sessions while process p is acquiring the lock. After p has inserted itself into Q, only processes that are ahead of p in Q can join their sessions before p does. Thus, the "time complexity" (as defined by Joung) of our algorithm using Lamport's or Mellor-Crummey and Scott's algorithm is $O(N)$.

Finally, we consider Joung's "concurrency" measure. For this measure, we consider a state in which a process p has "completed its request" for a session k and another process q is in session $h \neq k$. We count the worst-case number of processes that may join session h before q leaves session h. Using arguments similar to those above, it is straightforward to show that our algorithm yields $O(N)$ concurrency by this definition when used in conjunction with Lamport's or Mellor-Crummey and Scott's algorithm. Again, the algorithms of Yang and Anderson do not provide a bound for this measure. As Joung points out, higher concurrency can mean better resource utilization. For example, if more processes are able to read the CD currently loaded in the juke-box, we can avoid having to load it again later. Our concurrency measure is smaller than Joung's, which appears to be a disadvantage. However, it is easy to modify our algorithm to achieve higher concurrency if this is desirable with an additional shared variable, *Counter*. When a process is enqueued into an empty queue, it also sets *Counter* to zero. The test at line 4 is modified so that it is successful if $Session = t$ and either Q is empty or *Counter* is less than some threshold value. If this test is successful, then *Counter* is also increased at line 5. In this way, if there are still processes requesting the current session, they will not be blocked from entering until the desired number of processes have attended the session. In contrast, there is no obvious way to *reduce* the concurrency measure of Joung's algorithm. Thus, our algorithm is more flexible than Joung's with respect to this measure.

5 Eliminating Unnecessary Waiting

Clearly no (group) mutual exclusion algorithm can tolerate all process failures, because if a process fails in the critical section, then processes requesting other sessions can never enter their session without violating the mutual exclusion property. Therefore, it is common to assume, as we have, that processes do not fail outside their non-critical sections. However, in the group mutual exclusion problem, such assumptions can lead to unnecessary waiting. For example, in our algorithm, if one process is leaving a session, and therefore is holding the lock at statements 14 through 22, then another process that is trying to join the same session will be forced to wait, even though the problem definition does not *require* such waiting. It is therefore interesting to ask whether a solution can be found that forces processes to wait only when the problem definition requires it, without sacrificing the local spinning properties of our algorithm. Here we argue that this is possible in at least one scenario.

We first observe that the set S' of possible executions in which statements 4 through 10 and statements 14 through 21 are executed atomically is a subset of the set S of possible executions of our algorithm. (Note that in our algorithm, statements 1..2 and 12 can be executed while another process is executing in one of these two regions, so S' is a proper subset of S.) Thus, the properties we have proved of our algorithm would also hold if these two code regions were always executed atomically. Furthermore, if these regions were executed atomically, there would be no need to use mutual exclusion in our algorithm, and the unnecessary waiting caused by its use would be eliminated.

The effect of atomically executing the code regions mentioned above can be achieved by implementing them — as well as statements 1..2 and the test at statement 12 — as wait-free operations on a shared object consisting of the shared variables used in our algorithm. While this is a challenging problem to solve directly, there are a number of *universal constructions* (e.g., [1, 3, 11, 17]), which can be used to implement such operations. Furthermore, universal constructions exist in which read-only operations are implemented with only read operations (e.g., [3]). Thus, if we use such a construction to implement our group mutual exclusion algorithm as described above, it would still perform a bounded number of remote references in a cache-coherent system. We believe it would be possible to adapt this approach to achieve local spinning in a NUMA architecture too. However, there is little point in pursuing this direction before we determine whether the benefit of avoiding unnecessary waiting outweighs the additional overhead of the universal construction in the simpler case of a cache-coherent system.

The analysis shown in Table 1 for this approach assumes the use of the wait-free construction presented in [17][4]. This analysis is similar to that in the previous section, except that there is no need to consider what can happen while a process attempts to acquire the lock (because this approach does not use a lock). This accounts for the main difference in the analysis of this approach and using Mellor-Crummey and Scott's algorithm. That is, the problem of processes establishing and attending sessions while p is still trying to acquire the lock no longer exists. As a result, we have a stronger bound on the context switch complexity.

6 Concluding Remarks

We have presented a new solution to the group mutual exclusion problem recently posed by Joung [12]. This algorithm uses known solutions to the mutual exclusion problem, and its properties depend on the mutual exclusion algorithm used. We have discussed several possibilities for this choice, each of which has resulted in advantages over Joung's algorithm with respect to his complexity measures and/or with respect to local spinning in cache-coherent (CC) and non-uniform memory access (NUMA) systems. Our approach also yields constant time complexity in the absence of contention in some cases.

As discussed earlier, if the mutual exclusion algorithm chosen does not provide a bound on the number of times a process may be "overtaken" in the entry section, then our algorithm does not provide a bound on the complexity measures proposed by Joung. This limitation can be overcome through the use of a "helping" mechanism similar to the one used in [3]. The basic idea is as follows. Each time a process executes the entry section (of the group mutual exclusion algorithm), it also performs the same steps for another process that has previously announced its intention to enter a session, but has not yet acquired the critical section (of the mutual exclusion algorithm). Thus, with some minor changes to the mutual exclusion algorithm used, a process that is attempting to acquire the lock can occasionally check if it has been helped, and if so can give up on its attempt to acquire the lock. There are some subtleties to this approach, but all of these issues are addressed in [3], and the ideas would easily transfer into the context of this algorithm. Some of these issues are simplified by the fact that there can only be one process helping at any point in time.

Our use of known mutual exclusion algorithms results in an algorithm that is substantially simpler and more efficient than Joung's. Also, because our algorithm spins locally for some choices of the mutual exclusion algorithm, we expect that it will scale better than Joung's in a wide range of shared-memory

[4] The construction in [17], as presented there, does not have the property that read-only transactions are implemented with only read operations. However, for the simple kinds of transactions required here, it can easily modified so that it does.

multiprocessor systems. Furthermore, the simplicity of our algorithm has allowed us to produce simpler and more formal correctness proofs than Joung's.

Future work in this area includes implementations and performance studies, and also replacing the mutual exclusion algorithms with wait-free constructions to improve the complexity measures and eliminate unnecessary waiting.

Acknowledgements: The authors are grateful to Yuh-Jzer Joung for useful discussions.

References

[1] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pp. 538–547, 1995.

[2] Y. Afek, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. A bounded first-in, first-enabled solution to the ℓ-exclusion problem. *ACM Transactions on Programming Languages and Systems*, 16(3):939–953, 1994.

[3] J. Anderson and M. Moir. Universal constructions for large objects. In *Proceedings of the 9th Annual International Workshop on Distributed Algorithms*, pp. 168–182, 1995.

[4] J. Anderson and M. Moir. Using local-spin k-exclusion algorithms to improve wait-free object implementations. *Distributed Computing*, 11:1–20, 1997. A preliminary version appeared in *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, 1994, pp. 141-150.

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

[6] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, 1971.

[7] E. Dijkstra. Solution to a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[8] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Resource allocation with immunity to process failure. In *Proceedings of the 20th Annual IEEE Symposium on the Foundations of Computer Science*, pp. 234–254, 1979.

[9] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Distributed fifo allocation of identical resources using small shared space. *ACM Transactions on Programming Languages and Systems*, 11(1):90–114, 1989.

[10] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23:60–69, 1990.

[11] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

[12] Y.-J. Joung. Asynchronous group mutual exclusion. In *Proceedings of the 17th Annual ACM Symposium on the Principles of Distributed Computing*, pp. 51–60, 1998.

[13] P. Keane and M. Moir. A Simple Local-Spin Group Mutual Exclusion Algorithm. *Technical Report #TR99-04*, University of Pittsburgh, Department of Computer Science, 1999.

[14] L. Lamport. The mutual exclusion problem: Part II — statement and solutions. *Journal of the ACM*, 33(2):327–348, 1986.

[15] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

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

[17] M. Moir. Transparent support for wait-free transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, pp. 305–319, 1997.

[18] J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9:51–60, 1995.