Fast Local-Spin Abortable Mutual Exclusion with Bounded Space

Hyonho Lee

Department of Computer Science
University of Toronto, Toronto, ON, Canada, M5S 3G4
hlee@cs.toronto.edu

Abstract. Abortable mutual exclusion is a variant of mutual exclusion, where processes are allowed to abort their invocations while waiting to enter the critical section. In this paper, we present an FCFS abortable mutual exclusion algorithm with bounded time and space, in which each invocation performs $O(k^2)$ RMAs if at most k processes abort. We define an object type, *S-HAD*, from which it is easy to construct local-spin abortable mutual exclusion algorithms. Our main contribution is a wait-free implementation of an S-HAD object. We also develop a new, wait-free memory reclamation method, which generalizes reference counting, to achieve bounded space. The resulting algorithm uses $O(N^2)$ shared variables, each with $O(\log N)$ bits, where N is the number of processes.

1 Introduction

Abortable mutual exclusion [13] is a variant of classical mutual exclusion [5], in which a process performing a *trying protocol* to enter the *critical section* is allowed to stop waiting for the critical section to become available by performing an *abort protocol*, which returns the process to the *remainder section* within a bounded number of steps. Abortable mutual exclusion can be useful in real-time applications or in parallel database systems because, in these systems, users may want to abort any operation that takes too long [13].

In shared memory models, processes communicate with each other only via shared variables, so waiting processes must keep accessing shared variables until they stop waiting. Such *busy-waiting* may cause processes to perform an unbounded number of steps during the trying protocol. In the *distributed shared memory (DSM)* and *cache-coherent (CC)* models, the cost for a process to access its own local shared memory or cache is considered to be much less than the cost to access memory located remotely. Hence, in these models, counting only *remote memory accesses (RMAs, also known as remote memory references)* is a good measure of the time complexity of an algorithm. To achieve a bounded number of RMAs, many papers about mutual exclusion have considered *local-spin* algorithms. In such algorithms, each process accesses only a bounded number of RMAs while busy-waiting. In this paper, we restrict attention to local-spin algorithms.

C. Lu, T. Masuzawa, and M. Mosbah (Eds.): OPODIS 2010, LNCS 6490, pp. 364–379, 2010.
© Springer-Verlag Berlin Heidelberg 2010

In some classical mutual exclusion algorithms, such as the Bakery algorithm [10], which are not local-spin, all waiting processes wait for the same shared variable to change. Then a process can abort by simply announcing that it is no longer trying. Scott and Scherer [13] proposed two first-come-first-served (FCFS) local-spin mutual exclusion algorithms that allow waiting processes to abort. In their first algorithm, each process waits for a change in a certain shared variable associated with its predecessor (the last process that was enqueued before it). This algorithm is local-spin in the CC model. In their second algorithm, each waiting process first announces itself to its predecessor and then waits for a certain locally stored variable to change value. This algorithm is local-spin in the DSM model. In these algorithms, processes in the trying protocol form a queue and each waits for a signal from its predecessor. In Scott and Scherer's algorithms, each process enters the critical section within $O(1)$ RMAs when no process aborts. However, their abort protocol contains a waiting period in which an aborting process performs handshakes with its predecessor and successor in the queue, so it may not terminate the abort protocol within a bounded number of steps.

Later, Scott [12] eliminated this waiting period in the abort protocol: He presented two FCFS local-spin abortable mutual exclusion algorithms in which a process aborts within a bounded number of its own steps. When no processes abort, each invocation performs only a constant number of RMAs in the trying protocol. However, when two or more processes repeatedly abort without removing themselves from the queue of waiting processes and then re-enter the trying protocol, the length of the queue may become unbounded. Hence, these algorithms use unbounded space. The number of RMAs a process performs in the trying protocol can be as large as the number of consecutive times processes began the trying protocol immediately beforehand and subsequently aborted [12,9]. This can be arbitrarily large, since a process can repeatedly enter the trying protocol and abort. However, the bad situation is only achieved when each invocation that aborts decides to do so before its predecessor begins the abort protocol.

In Section 2.4 of [12], Scott described a simple abortable mutual exclusion algorithm with $\Theta(N)$ space. This algorithm also uses a queue. When a process starts its trying protocol, it enqueues an element, and waits for the value of its predecessor in the queue to change. When a process aborts, it changes the value of the element it last enqueued. If this process re-enters the trying protocol, then it checks whether the element it last enqueued has been accessed and, if not, it reclaims this element, instead of enqueuing a new element. However, in this algorithm, a process can perform an unbounded number of RMAs in the trying protocol. For example, suppose process p is the predecessor of another process q in the queue, and q is waiting for the value of p's element to be changed. When p aborts, it changes the value of its element. If it re-enters the trying protocol and reclaims the same element, it changes the value of the element back to its previous value. Even if q did not notice p's abort, q's next read of p's element generates a cache miss. Thus, if p aborts and re-enters the trying

protocol, reclaiming the same element an unbounded number of times, q may perform an unbounded number of RMAs while waiting. Thus this algorithm is not local-spin or FCFS.

There are two previously known FCFS local-spin abortable mutual exclusion algorithms with bounded space in which each process performs a bounded number of RMAs for each entry to the critical section: Jayanti [9] uses registers and LoadLinked/StoreConditional (LL/SC), and Danek and Lee [3] use only registers. Jayanti's algorithm performs $\Theta(\min(k, \log N))$ and Danek and Lee's algorithm performs $\Theta(N)$ RMAs for each entry to the critical section, where N is the number of processes and k is the contention, i.e., the number of processes that are trying to enter the critical section at the same time. Danek and Lee also presented a local-spin abortable mutual exclusion algorithm with $\Theta(\log N)$ RMAs that does not satisfy FCFS. Since any mutual exclusion algorithm using only registers and comparison primitives, such as COMPARE_AND_SWAP or LL/SC, requires $\Omega(\log N)$ RMAs in the worst case for each entry to the critical section [2] and, since mutual exclusion is a special case of abortable mutual exclusion, both Jayanti's algorithm and Danek and Lee's $\Theta(\log N)$ algorithm are optimal.

In the worst case, each process performs fewer RMAs in Jayanti's algorithm than in Scott's local-spin algorithms. However, if the number of consecutive aborts is $o(\log N)$, then Scott's algorithms are better in terms of the number of RMAs. A natural question is whether there exists a local-spin abortable mutual exclusion algorithm that preserves all of the merits of Scott's algorithms, but uses only bounded space and performs a bounded number of RMAs in the worst case.

In this paper, we present a new FCFS local-spin abortable mutual exclusion algorithm for the CC model. It uses $O(N^2)$ space and a process performs $O(k^2)$ RMAs to enter the critical section, where k is the number of processes that began the trying protocol immediately beforehand and subsequently aborted. The worst case is only achieved when each invocation returns to the remainder section and re-enters the trying protocol before its predecessor begins the abort protocol.

For modularity, we first define an object type, *S-HAD*, from which it is easy to construct a local-spin abortable mutual exclusion algorithm. S-HAD is a sequence that supports Head, Append, and Delete, but with two restrictions: Each process can own at most one element in the sequence at a time and only the owner of an element can perform these three operations on it.

We give two wait-free implementations of an S-HAD object. Our first implementation has $O(N^2)$ RMA complexity but uses unbounded space. Then, we extend it, using a generalization of reference counts, to achieve $O(N^2)$ space complexity as well. Our new memory reclamation method is wait-free and very efficient in terms of RMAs. It uses only standard operations (TEST_AND_SET, FETCH_AND_ADD, FETCH_AND_STORE, READ and WRITE) on $O(\log N)$ bit words, and each process performs $O(1)$ RMAs for recycling a record. In contrast, Detlefs et al.'s reference counting method [4] uses DOUBLE_COMPARE_AND_SWAP, which is not available in most systems, and Valois's reference counting method [14]

allows processes to access a freed record or a recycled record, which would cause a significant increase in the RMA complexity of our algorithm. With *hazard pointers* [11], to reuse a record, a process must read the hazard pointers of all other processes, which takes $\Theta(N)$ RMAs. Herlihy et al. [7] proposed a reference counting method similar to hazard pointers. Their method also takes $\Theta(N)$ RMAs. Since we want each process to perform a small number of RMAs if aborts are rare, we needed to develop a new memory reclamation method.

Section 2 formally defines abortable mutual exclusion and describes the system model. Section 3 defines S-HAD, gives an abortable mutual exclusion algorithm based on S-HAD, and proves the correctness of the algorithm. Section 4 presents our unbounded space implementation of S-HAD, and Section 5 presents our bounded space implementation of S-HAD. Complete proofs of correctness of the algorithms in Sections 4 and 5 appear in the full paper.

2 Preliminaries

In an abortable mutual exclusion algorithm, processes that want to access the critical section first execute the trying protocol. After completing the trying protocol, a process enters the critical section. When it finishes the critical section, it then performs the *exit protocol*, and finally returns to the remainder section. If a process must wait in the trying protocol and wants to abort, it performs the abort protocol, and then returns to the remainder section. We assume no process failures.

An algorithm solves the abortable mutual exclusion problem, if it satisfies the following properties:

Mutual Exclusion: At most one process is in the critical section at any time.
Lockout Freedom: If a process p starts executing the trying protocol and keeps taking steps in the trying protocol without aborting, then it will eventually enter the critical section.
Bounded Exit: If a process starts executing the exit protocol, then it returns to the remainder section within a bounded number of its own steps.
Bounded Abort: If a process starts executing the abort protocol, then it returns to the remainder section within a bounded number of its own steps.

The *First-Come-First-Served (FCFS)* property [10] is a strong fairness condition in which processes enter the critical section in roughly the same order they enter the trying protocol. Although it is not a requirement of abortable mutual exclusion, most mutual exclusion algorithms in which each process performs $O(1)$ RMAs to enter the critical section satisfy this property.
FCFS: The doorway is a bounded section of code that begins the trying protocol. If a process p finishes executing the doorway before a process q begins executing the doorway, and p does not abort, then p enters the critical section before q does.

In this paper, we consider the asynchronous cache-coherent (CC) model with N processes [1]. The CC model is a shared memory model in which each process

has its own local cache. In this model, processes perform atomic operations on shared variables. We divide all atomic operations into two classes: *trivial* operations, which cannot change the value of a shared variable, and *non-trivial* operations, which may change the value of a shared variable. READ is an example of a trivial operation. WRITE, FETCH_AND_STORE and COMPARE_AND_SWAP are examples of non-trivial operations.

When a process p performs a trivial operation on a shared variable, it first checks its own cache. If p has a valid cached copy of the shared variable (i.e. no other process has performed a non-trivial operation on the shared variable since p last accessed the shared variable and copied it to its cache), the trivial operation does not generate an RMA. If p does not have a valid cached copy of the shared variable (either because p has not accessed the shared variable before or because another process has performed a non-trivial operation on the shared variable after p's last access of the shared variable), then p accesses the shared variable from remotely located shared memory and copies the variable to its own cache. This generates an RMA. When p performs a non-trivial operation on a shared variable, even if the value of the variable does not change, the system invalidates all other cached copies of the variable, which generates an RMA.

A *passage* is the sequence of steps performed by a process from when it begins the trying protocol until it next returns to the remainder section by finishing the exit or abort protocol. Our complexity measure is the worst case number of RMAs performed in the trying, exit, and abort protocols in any passage.

3 S-HAD and Abortable Mutual Exclusion

An S-HAD is a sequence of elements, each owned by a different process. A process can perform the following operations on an element that it owns:

Head(R): returns TRUE if element R is at the beginning of the sequence.
Append(R): appends element R to the end of the sequence.
Delete(R): deletes element R from the sequence.

Append(R) may be called only when R is not in the sequence, and Delete(R) may be called only when R is in the sequence. Thus, element R occurs in the sequence if and only if Delete(R) has not been performed since Append(R) was last performed. Head(R) is TRUE if and only if R occurs in the sequence and each element X that was appended before R has been deleted from the sequence.

We can easily build an abortable mutual exclusion algorithm using a linearizable implementation of an S-HAD object. When a process tries to enter the critical section, it appends a new element to the S-HAD object. Then the process keeps performing Head until its element is at the head of the S-HAD object. When Head returns TRUE, the process enters the critical section. When the process finishes the critical section or wants to abort, it deletes the appended element from the S-HAD object. The detailed algorithms TryingProtocol, ExitProtocol and AbortProtocol appear in Figure 1. GetNewElement is a function that returns a new element. This may be a system call that allocates a memory location for an element or a function that returns an element from a free list.

```
TryingProtocol()
  T1: R := GetNewElement()
  T2: Append(R)
  T3: while ¬ Head(R) do
  T4:     if the process wants to abort, perform AbortProtocol()
      end while

ExitProtocol() / AbortProtocol()
  E1: Delete(R)
```

Fig. 1. Abortable Mutual Exclusion Algorithm

To prove the correctness of this abortable mutual exclusion algorithm, we show that only the process whose element is at the head of the sequence enters the critical section. We also show that any appended element eventually becomes the head of the sequence if it is not deleted. If process p gets an element R on line T1, we say $owner(R) = p$.

Observation 1. *If a process p is in the critical section, then the element R at the head of the sequence is owned by p.*

An operation is *wait-free* if a process performs the operation within a bounded number of its own steps. Since any abortable mutual exclusion algorithm must satisfy the bounded exit and bounded abort properties, Delete must be wait-free. If GetNewElement, Append and Head are also wait-free, then the while loop starting on line T3 is the only waiting period. In this case, the algorithm in Figure 1 is an FCFS abortable mutual exclusion algorithm.

Theorem 1. *Given wait-free implementations of an S-HAD object and GetNewElement, the algorithm in Figure 1 is an FCFS abortable mutual exclusion algorithm.*

Proof. The mutual exclusion property follows from Observation 1. Since Delete is wait-free, the algorithm satisfies the bounded abort and bounded exit properties. To prove lockout freedom, suppose that there exists an infinite execution E in which some set of processes, $\mathcal{P}$, keep performing TryingProtocol without entering the critical section or performing AbortProtocol.

Since GetNewElement and Append are wait-free, each process p in $\mathcal{P}$ eventually gets a new element, R_p, on line T1 of its last invocation of TryingProtocol, and finishes performing Append(R_p) on line T2. Since p does not perform ExitProtocol or AbortProtocol after its last invocation of TryingProtocol, p does not subsequently perform Delete(R_p). Let $\mathcal{R} = \{R_p | p \in \mathcal{P}\}$. Let X be the element in $\mathcal{R}$ appended earliest, and let $p \in \mathcal{P}$ be the process that performed Append(X).

By definition, any invocation that last appended an element S before X either eventually enters the critical section and performs ExitProtocol, or eventually performs AbortProtocol. Hence, the invocation eventually performs Delete(S). Thus, eventually, X becomes the head of the sequence and Head(X) returns

TRUE. Since p keeps performing TryingProtocol without performing AbortProtocol, it performs Head(X) infinitely many times. Thus, p will eventually enter the critical section. This contradicts the assumption that $p \in \mathcal{P}$, so the algorithm satisfies lockout freedom.

Since GetNewElement and Append are wait-free, each process performs lines T1 and T2 within a bounded number of its own steps. Let the doorway be lines T1 and T2. If process p finishes Append(R) before process q starts an invocation of GetNewElement that returns R', then R is appended before R'. Thus, if p does not abort, R reaches the head of the sequence before R'. Then, by Observation 1, p enters the critical section before q. Hence, the resulting abortable mutual exclusion algorithm satisfies the FCFS property. □

In some systems, allocating a memory location may not be wait-free. However, the algorithm in Figure 1 still solves abortable mutual exclusion if GetNewElement satisfies the following properties: a process that invokes GetNewElement eventually completes GetNewElement and a process that invokes GetNewElement but wants to return to the remainder section before it completes can do so within a bounded number of its own steps. These properties are required for lockout freedom and bounded abort, respectively.

For this algorithm to be local-spin, Head must be implemented carefully. The RMA complexity of one passage is the sum of the RMAs performed during one execution of each of GetNewElement, Append and Delete, and an unbounded number of executions of Head. Thus, in the DSM model, if Head contains even a single RMA, then the resulting algorithm is not local-spin. However, in the CC model, when a process reads a shared variable, it copies its value to its local cache. Hence, even if Head contains remote memory reads, subsequent calls of Head by process p do not generate RMAs unless another process performs a non-trivial operation on a shared variable p reads in Head.

In the next two sections, we present wait-free, linearizable implementations of an S-HAD object shared by N processes such that any number of calls of Head(R) between a call of Append(R) and the subsequent call of Delete(R) generate only a bounded number of RMAs in the CC model. Moreover, if each element is deleted only when it is at the head of the sequence, this number of RMAs is bounded above by a small constant. Our first implementation is simpler but uses unbounded space, and our second implementation uses bounded space.

4 A Simple Implementation of S-HAD

In this section, we present a simple implementation of an S-HAD object. Detailed pseudo-code is given in Figure 2. Note that the lines are not consecutively numbered. This is so each line has the same number as in the bounded space implementation in Section 5.

We begin by explaining the overall structure of the implementation. An S-HAD object is represented by an intree of records, one per element, each with a pointer, *pred*, which is either NIL or points to another record, and a flag, *del*. The root of the tree is a *dummy* record, which is never deleted, whose *del* field is

always 'head' and whose *pred* field is always NIL. For every other record R, the field $R.del$ indicates whether the element it represents is in the S-HAD sequence or has been *logically deleted*. The initial value of $R.del$ is FALSE, and it becomes TRUE when the owner of R performs line D1 of Delete(R). The field $R.pred$ points to another record that was appended before R. Thus, the records form an acyclic graph rooted at the dummy record. There is a FETCH_AND_STORE (or SWAP) object, $Tail$, that initially points to the root. To perform Append(R), a process atomically reads $Tail$ and updates $Tail$ to point to R on line A2 of Append(R). Hence, $Tail$ always points to the record that was appended most recently.

When a process wants to know whether the element represented by its record R is at the head of the sequence, it repeatedly updates $R.pred$ until it points to a record that has not been deleted. This is done by Update(R). Then, the element represented by R is at the head of the sequence if and only if $R.pred$ points to the dummy record.

A process logically deletes its record R by setting $R.del$ to TRUE. Then it calls Update(R) one more time to ensure that R does not point to another logically deleted record. This is necessary because, otherwise, a sequence with two records that are preceded by arbitrarily many logically deleted records between them and the dummy record can be created by repeatedly deleting the second last record and then appending a new record.

At any point during an execution, the state of the S-HAD object is the sequence of records R for which line A2 of Append(R) has been performed and $R.del$ = FALSE. This sequence is ordered by the time at which line A2 of Append(R) was performed. All records that represent elements in the S-HAD sequence are on the same path to the root and the one that is closest to the root is at the head of the sequence.

We define the linearization point of Append(R) to be when line A2 is performed. Immediately afterwards, $R.del$ = FALSE. Hence, by performing line A2 of Append(R), the element represented by R is appended to the end of the sequence. The element represented by R is removed from the sequence when $R.del$ is set to TRUE on line D1. We define this to be the linearization point of Delete. We define the linearization point of Head(R) to be when Update(R) returns on line H1, which is when line U2 of Update(R) is performed with $(*mypred).del \neq$ TRUE. The correctness of the implementation in Figure 2 follows from the next two results.

Observation 2. *At the linearization point of Head(R), let S be the record pointed to by $R.pred$ and let $d = S.del$. Then Head(R) returns* TRUE *if and only if S is the dummy record. If Head(R) returns* FALSE*, then d =* FALSE*.*

Lemma 3. *Head(R) returns* TRUE *if and only if the element represented by R is at the head of the sequence at the linearization point of Head(R).*

Append(R) is wait-free, since it consists of only two atomic operations. Similarly, Head(R) and Delete(R) are wait-free if Update(R) is wait-free. The following lemma shows that Update is wait-free.

```
shared variables:
    type Record ( pred: pointer to a record ∪ { NIL }, initially NIL
                  del: { TRUE, FALSE, 'head' }, initially FALSE)
    Record Dummy = (NIL, 'head')
    Tail: pointer to a record, initially points to Dummy
private variables:
    mypred, ppred: pointer to a record

Head(R :Record) % Precondition: R.del = FALSE, R.pred ≠ NIL
% Postcondition: returns TRUE, if R is the head of the list; otherwise, returns FALSE
  H1: Update(R)
  H2: mypred := R.pred
  H3: return ((*mypred).del = 'head')

Append(R :Record) % Precondition: R.del = FALSE, R.pred = NIL
  A2: mypred := FETCH_AND_STORE(Tail, &R)
  A3: R.pred := mypred

Delete(R :Record) % Precondition: R.del = FALSE, R.pred ≠ NIL
  D1: R.del := TRUE
  D2: Update(R)

Update(R :Record) % Precondition: R.pred ≠ NIL
  U1: mypred := R.pred
  U2: while (*mypred).del = TRUE do
  U3:     ppred := (*mypred).pred
  U5:     R.pred := ppred
  U9:     mypred := ppred
      end while
```

Fig. 2. An Implementation of S-HAD

Lemma 4. *If no record is appended more than once, then the while loop of Update(R) is not performed forever.*

Proof sketch. In each execution of the while loop in Update(R), $R.pred$ is updated. Each time $R.pred$ is updated, $R.pred$ points to a record that was appended earlier than the record it previously pointed to. Since the number of records that were appended earlier than R is bounded, $R.pred$ is updated a bounded number of times. □

Hence, the implementation in Figure 2 is wait-free. If GetNewElement is a wait-free system call that always returns a new record, then, by Theorem 1, the algorithm in Figure 1 using the implementation in Figure 2 is a correct FCFS abortable mutual exclusion algorithm.

While $R.pred$ does not change, any sequence of calls to Head(R) generates at most three RMAs in the CC model: the first time $owner(R)$ reads $R.pred$ and $(*R.pred).del$, and when $(*R.pred).del$ changes from FALSE to TRUE. If

all records are deleted in the same order as they are appended, which is the case for our abortable mutual exclusion when no aborts occur, then *R.pred* changes only once. Hence, in the abortable mutual exclusion algorithm using this implementation of S-HAD, each process performs $O(1)$ RMAs if no aborts occur.

We say that a record R was *deleted prematurely* if *R.pred* did not point to the dummy record when line D1 of Delete(R) was performed. In the abortable mutual exclusion algorithm in Figure 1, each invocation that is aborted corresponds to a prematurely deleted record. If k is the number of processes that delete records prematurely, then we prove that the *pred* pointer of every record changes $O(k^2)$ times.

Lemma 5. *Let R' be the last record that was appended prior to element R, but was not prematurely deleted. If k' is the number of different processes that appended records between R' and R inclusive, then the while loop of Update(R) was performed at most $k'(k'+3)/2$ times between beginning Append(R) and completing Delete(R).*

Proof sketch. A record X is *active* if and only if the first line of Append(X) has been performed, but the last line of Delete(X) has not yet been performed. Note that, if the element represented by X is in the sequence, then X is also active, but the converse may not hold after line D1 of Delete(X) has been performed. After record X becomes inactive, *X.pred* does not change.

Suppose there is a sequence of records, $W_1, W_2, \ldots, W_{j-1}, W_j$ such that W_{i+1}*.pred* points to W_i for $1 \leq i < j$. In this case, we say that there is a *path* from W_j to W_1. If all of $W_1, \ldots, W_j$ are inactive, then W_i was active when W_{i+1} became inactive. Hence, Delete(W_{i+1}) was completed before Delete(W_i). In particular, Delete(W_j) was completed before Delete(W_1). If $owner(W_1) = owner(W_j)$, then Delete(W_1) was completed before W_j was appended and hence, before Delete(W_j) was completed. Thus, in this case, at least one of $W_1, \ldots, W_{j-1}$ is active.

Since there are k' different processes that appended an element between R' and R inclusive during Update(R), the path from R to R' contains $\ell \leq k'$ active records, the first of which is R. If Y_i is the ith active record between R' and R, for $i = 1, \ldots, \ell$, then the records between Y_i and Y_{i+1} are all inactive. It follows from the previous paragraph that each inactive record on the path from Y_{i+1} to Y_i has a different owner. Hence, the path from Y_{i+1} to Y_i contains at most $k' - i$ records. Also, the subpath from Y_1 to R' contains at most k' records. Thus, the number of records on any path from R to R' is at most $(k'-\ell+1)+(k'-\ell+2)+\ldots+(k'-1)+k'+\ell = \ell(2k'-\ell+3)/2 \leq k'(k'+3)/2$. Since R' was not prematurely deleted, R'*.pred* pointed to the dummy record when R'*.del* was set to TRUE. Hence, immediately after R' was logically deleted, the path from R to the dummy record contained at most $k'(k'+3)/2+1$ records. Therefore, *R.pred* was updated at most $k'(k'+3)/2$ times. □

Excluding Update, each process performs $O(1)$ RMAs during Append, Head, and Delete. Lemma 5 implies that a process performs $O(k^2)$ RMAs between

beginning Append(R) and completing Delete(R). Thus, if GetNewElement takes $O(1)$ RMAs, the algorithm in Figure 1 using the implementation in Figure 2 has $O(k^2)$ RMA complexity.

Theorem 2. *Suppose GetNewElement takes $O(1)$ RMAs. In the algorithm in Figure 1 using the implementation in Figure 2, each process performs $O(k^2)$ RMAs per passage, where k is the number of processes that began the trying protocol immediately beforehand and subsequently aborted.*

Since k is bounded by N, the worst case RMA complexity is $O(N^2)$. This worst case can occur, but only if $\Theta(N)$ processes perform particular sequences of Appends and Deletes. A specific execution that generates the worst case is described in the full paper.

5 An Implementation of S-HAD with Bounded Space

In the previous algorithm, even though records have been logically deleted from the S-HAD object, processes can still access them to find out that they have been deleted. Also, each time a process performs Append, it uses a new record. Because logically deleted records are not deallocated, that algorithm uses unbounded space. However, eventually, a logically deleted record is no longer accessed, and we can safely reclaim the memory used by that record. To determine when a logically deleted record is no longer accessed, we use a generalization of *reference counts*. If the generalized reference count for a record becomes zero, then the record can be physically deleted, since no process will subsequently access the information in the record.

In simple reference counting, each record contains a counter and a record can be physically deleted when its counter is zero. If record R points to record X, record S points to Y, and a process wants to change R to point to the same place as S, then it reads $\&Y$ from S, increments Y's counter, sets R to $\&Y$, and finally decrements X's counter. However, if Y's counter becomes zero and Y is physically deleted between the first two steps, then the owner of R will not notice this and may access the location in memory from which Y was deleted. One way to prevent this is to perform the first two steps atomically using DOUBLE_COMPARE_AND_SWAP (Detlefs et al. [4]). Unfortunately, DOUBLE_COMPARE_AND_SWAP is not available in most systems.

Another approach is for the process to read S after it increments Y's counter and, if S's pointer has been changed, it decrements Y's counter instead of changing R [14]. In this case, no other information in Y is accessed. However, this method still allows access to the counter of a physically deleted record, so the memory it occupies cannot be reclaimed by the system. Physically deleted records can be put into a free list and reused in the future. However, when processes access the counter of a free or recycled record, extra RMAs are generated. Using this method in our algorithm increases the worst case RMA complexity from $\Theta(N^2)$ to $\Theta(N^4)$.

```
shared variables:
    type Record ( rc: a pair of integers (orc, drc), where 0 ≤ orc < N and
                      −N < drc < N, initially (0, 0)
                  pred: a pair (predptr, prc), where predptr is NIL or a pointer to
                      a record and 0 ≤ prc < N is an integer, initially (NIL, 0)
                  del: { TRUE, FALSE, 'head' }, initially FALSE
                  done: { TRUE, FALSE}, initially FALSE)
    Record Dummy = ((0,0), (NIL, 0), 'head', 0)
    Tail: pointer to a record, initially points to Dummy
private variables:
    mypred, ppred: pointer to a record
    myprc, x, y: integer

Head(R :Record) % Precondition: R.del = FALSE, R.pred ≠ (NIL, −)
% Postcondition: returns TRUE, if R is the head of the list; otherwise, returns FALSE
  H1: Update(R)
  H2: (mypred, −) := R.pred
  H3: return ((*mypred).del = 'head')
Append(R :Record) % Precondition: R.del = FALSE, R.pred = (NIL, 0)
  A1: R.rc := (1, 1)
  A2: mypred := FETCH_AND_STORE(Tail, &R)
  A3: R.pred := (mypred, 0)
Delete(R :Record) % Precondition: R.del = FALSE, R.pred ≠ (NIL, −)
  D1: R.del := TRUE
  D2: Update(R)
  D3: Remove(R)
Update(R :Record) % Precondition: R.pred ≠ (NIL, −)
  U1: (mypred, −) := R.pred
  U2: while (*mypred).del = TRUE do
  U3:   (ppred, −) := FETCH_AND_ADD((*mypred).pred, (0, 1))
  U4:   FETCH_AND_ADD((*ppred).rc, (1, 0))
  U5:   (−, myprc) := FETCH_AND_STORE(R.pred, (ppred, 0))
  U6:   (x, y) := FETCH_AND_ADD((*mypred).rc, (−1, myprc − 1))
  U7:   if (x, y) = (1, 1 − myprc) then
        % Note that (*mypred).rc = (0, 0)
  U8:     Remove(*mypred)
        end if
  U9:   mypred := ppred
      end while
Remove(R :Record)
  R1: if TEST_AND_SET(R.done) = TRUE then
  R2:   (mypred, myprc) := FETCH_AND_STORE(R.pred, (NIL, 0))
  R3:   (x, y) := FETCH_AND_ADD((*mypred).rc, (−1, myprc − 1))
  R4:   recycle(R)
  R5:   if (x, y) = (1, 1 − myprc) then
        % Note that (*mypred).rc = (0, 0)
  R6:     Remove(*mypred)
        end if
      end if
```

Fig. 3. An Implementation of S-HAD with bounded space

Due to their weaknesses, instead of adopting previous methods, we devise a new reference counting method for our algorithm. In our new memory reclamation method, each record has a pointer *predptr*, and an *original reference counter* (*orc*), which stores an upper bound on the number of pointers in shared memory that point to it. In addition to *orc*, each record also has two more counters, a *proactive reference counter* (*prc*) and a *distributed reference counter* (*drc*). Both *prc* and *drc* are used to keep track of pointers that have been read and may be written to shared memory in the future.

R.prc stores the number of times *R.predptr* has been read since *R.predptr* was last updated. This value is transferred to *S.drc* when *R.predptr* is changed from pointing to S to pointing to another record. In general, for any record S, the sum of the *prc*'s of all records that point to S plus *S.drc* is bounded above by the number of times a pointer to S has been read minus the number of times a pointer to S has been overwritten.

R.drc is stored together with *R.orc* in a single variable *R.rc*, so that they can be accessed together. The range of *orc* is from 0 to $N-1$ and the range of *drc* is from $1-N$ to $N-1$. Hence, $rc = (drc,orc)$ can be represented using $O(\log N)$ bits in a single word of memory. FETCH_AND_ADD$(rc,(m,n))$ can be simulated by FETCH_AND_ADD$(rc, m \cdot 2^{\lceil \log_2 N \rceil} + n)$.

R.prc is stored together with *R.predptr* in a single variable *R.pred*. Associating a pointer with a counter was also done in [6] and [8]. In [8], a wait-free implementation of a pointer requires a complicated atomic operation. However, in our algorithm, processes perform only READ, WRITE, FETCH_AND_ADD and FETCH_AND_STORE operations on pointers. Pointers in [6] are similar to ours, but are stored together with two integers.

Since the range of *prc* is from 0 to $N-1$, *pred* can be represented using $\lceil \log_2 N \rceil$ bits in addition to the bits used for the pointer, all stored in one word. Since we use only $O(N^2)$ records, a pointer can be represented using $O(\log N)$ bits. FETCH_AND_ADD$(pred,k)$ adds k to *prc*. In our algorithm, whenever *predptr* is set to point to a record X, *prc* becomes zero, which can be accomplished by FETCH_AND_STORE$(pred,(\&X,0))$.

Pseudo-code for the algorithm is presented in Figure 3. Head(R) is essentially the same as in the previous algorithm. In Append(R), the owner of R sets *R.rc* to (1,1) before appending R to the end of the sequence. Most of the differences are inside the while loop of Update(R). Unlike the previous algorithm, *R.pred* can now be changed by processes other than the owner of R, on lines U3 and R2, but only after R has been logically deleted. This does not affect the RMA complexity of Head(R), which is only performed while R is in the sequence.

To see how Update(R) was modified, consider the situation when process p, which owns record R, wants to update R's predecessor pointer to point to the predecessor of its predecessor, i.e. *R.predptr* := (**R.predptr*).*predptr*. Suppose X is R's predecessor, Y is X's predecessor, and p's local variable *mypred* points to X. To change R to point from X to Y, process p performs line U3, in which p atomically reads *X.predptr* and increments *X.prc* using FETCH_AND_ADD. This indicates that R will reference Y and it learned about Y from X. Next, p increments

$Y.orc$ on line U4. On line U5, p atomically changes $R.predptr$ to Y, reads $R.prc$ into its local variable $myprc$, and resets $R.prc$ to 0, using FETCH_AND_STORE. Hence, $myprc$ stores the number of processes that have accessed $R.predptr$ between the last two updates of $R.predptr$. Finally, on line U6, p atomically decrements $X.orc$ and adds $myprc - 1$ to $X.drc$, using FETCH_AND_ADD. The distributed reference count is decremented, since R is no longer pointing to X. The value that had been stored in $R.prc$ before it was reset is transferred to $X.drc$. Both of these are accomplished by adding $myprc - 1$ to $X.drc$.

When a process tries to physically delete a record R, it calls function Remove(R). R can be physically deleted only when no record points to R, no records will point to R, and lines D1 and D2 of Delete(R) have been completed. $R.orc = 0$ indicates that no record currently points to R, and $R.drc = 0$ indicates that no record will point to R. Hence, when $R.rc = (0, 0)$ and Delete(R) is completed, R can be physically deleted. To ensure that both conditions are met, Remove(R) is called twice: one by the owner of R at the end of Delete(R) (line D3) and the other by a process who finds that $R.rc = (0, 0)$ during Update (line U8) or Remove (line R6). Only the later of these two calls physically deletes R by calling recycle(R) on line R4. recycle(R) can be either a system call that deallocates R from memory or some function that moves R into a free list.

Remove(R) is called from exactly one of line U8 or line R6, so Remove(R) is called exactly twice. To ensure that only the later call physically deletes R, we use a TEST_AND_SET object, $R.done$, and only perform the rest of Remove if it returns TRUE. Note that a record R can be physically deleted by any process, although Delete(R) can be called only by the owner of R.

Remove is called recursively if physically deleting a record causes another record's reference counts to become $(0, 0)$. When a process physically deletes a record R, it also removes its pointer, $R.predptr$. If $R.predptr$ pointed to another record S, then S's reference counts must be updated. This may cause $S.rc$ to become $(0, 0)$ and, if it is, Remove(S) is called recursively on line R6. These recursive calls add only $O(k^2)$ RMAs in total, if k is the number of processes that appended a record before R and deleted it prematurely. Hence, it does not affect the overall asymptotic RMA complexity of the algorithm.

Unlike the reference counting in [14], our algorithm allows each record to be reclaimed by the system, provided the system calls for memory allocation and deallocation each take $O(1)$ RMAs. In this case, GetNewRecord in Figure 1 is a system call for memory allocation and recycle(R) on line R4 of Figure 3 is a system call for memory deallocation.

Alternatively, we can use a free list of length at most $3N$ for each process. The reason $3N$ records per process suffice is discussed in the full paper. Each process, p, maintains a Boolean array of size $3N$, which indicates which records are available. To get a new record, process p keeps checking each element of the array until it finds a true bit. If the ith bit in the array is TRUE, p sets it to FALSE and uses its ith record. When some process recycles the ith record of p, it sets the ith element of p's array to TRUE. Since p is the only process that sets

Table 1. Local-spin abortable mutual exclusion algorithms

	Scott [12]	Jayanti [9]	Danek and Lee [3]	New Algorithm
Atomic operations used besides READ and WRITE	FETCH_AND_STORE, COMPARE_AND_SWAP	LL/SC	-	TEST_AND_SET, FETCH_AND_ADD, FETCH_AND_STORE
Local-spin on CC	Yes	Yes	Yes	Yes
Local-spin on DSM	Yes	Yes	Yes	No
RMAs / passage if no aborts	$O(1)$	$\Theta(\log N)$	$\Theta(\log N)$; $\Theta(N)$	$O(1)$
RMAs / passage	unbounded	$\Theta(\log N)$	$\Theta(\log N)$; $\Theta(N)$	$O(N^2)$
space	unbounded	$\Theta(N)$	$\Theta(N)$	$\Theta(N^2)$
FCFS	Yes	Yes	No ; Yes	Yes

elements of its array to FALSE, no RMA is generated when p reads FALSE. Therefore, both GetNewRecord in Figure 1 and recycle(R) on line R4 of Figure 3 generate only $O(1)$ RMAs.

The resulting algorithm uses only $O(N^2)$ space. It also has the same RMA complexity, $O(N^2)$, as the previous algorithm. Therefore, the abortable mutual exclusion algorithm in Figure 1 using this implementation of S-HAD is local-spin, uses $O(N^2)$ space, has $O(N^2)$ RMA complexity, and each process performs $O(k^2)$ RMAs per passage, where k is the number of processes that began the trying protocol immediately beforehand and subsequently aborted.

6 Conclusions

We presented a local-spin abortable mutual exclusion algorithm with $O(N^2)$ space, in which each process performs $O(1)$ RMAs for each entry to the critical section when no processes abort, and each process performs $O(k^2)$ RMAs when aborts occur in the CC model, where k is the number of processes that abort. Table 1 compares our algorithm with previous abortable mutual exclusion algorithms.

Our algorithm performs more RMAs per passage than Jayanti's and Danek and Lee's in the worst case, but fewer when no aborts occur. If $k = o(\sqrt{\log N})$ processes began the trying protocol immediately before process p and subsequently aborted, then p performs $o(\log N)$ RMAs per passage in our algorithm, which is better than Jayanti's or Danek and Lee's algorithms. It would be interesting to compare the experimental performance of our algorithm with the other algorithms.

It is open whether $\Omega(N^2)$ space and RMAs are necessary in the CC model, if each process performs a constant number of RMAs when no processes abort. It is also open whether there exists a local-spin abortable mutual exclusion algorithm in the DSM model with bounded space and RMAs, in which each process performs a constant number of RMAs when no processes abort.

Acknowledgements

I would like to thank Professor Faith Ellen and the anonymous reviewers for numerous helpful suggestions and careful corrections.

References

1. Anderson, J.H., Kim, Y.-J., Herman, T.: Shared-Memory Mutual Exclusion: Major Research Trends Since 1986. Distributed Computing (2002)
2. Attiya, H., Hendler, D., Woelfel, P.: Tight RMR Lower Bounds for Mutual Exclusion and Other Problems. In: Proceedings of the 40th Annual ACM Symposium on Theory of Computing, pp. 217–226 (2008)
3. Danek, R., Lee, H.: Brief Announcement: Local-Spin Algorithms for Abortable Mutual Exclusion and Related Problems. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 512–513. Springer, Heidelberg (2008)
4. Detlefs, D.L., Martin, P.A., Moir, M., Steele Jr., G.L.: Lock-Free Reference Counting. In: The 20th Annual ACM Symposium on Principles of Distributed Computing, pp. 190–199 (2001)
5. Dijkstra, E.W.: Solution of a Problem in Concurrent Programming Control. Communications of the ACM 8(9), 569 (1965)
6. Goldberg, B.: Generational Reference Counting: A Reduced-communication Distributed Storage Reclamation Scheme. In: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation (1989)
7. Herlihy, M., Luchangco, V., Martin, P., Moir, M.: Brief Announcement: Dynamic-sized lock-free data structures. In: Proceedings of the 21st Annual Symposium on Principles of Distributed Computing (2002)
8. Herlihy, M., Luchangco, V., Moir, M.: Space and Time Adaptive Non-blocking Algorithms. Electronic Notes in Theoretical Computer Science 78, 260–280 (2003)
9. Jayanti, P.: Adaptive and Efficient Abortable Mutual Exclusion. In: Proceedings of the 22th Annual ACM Symposium on Principles of Distributed Computing (July 2003)
10. Lamport, L.: A New Solution of Dijkstra's Concurrent Programming Problem. Communications of the ACM 17(8), 453–455 (1974)
11. Michael, M.M.: Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. IEEE Transactions on Parallel and Distributed Systems 15(6) (2004)
12. Scott, M.L.: Non-blocking Timeout in Scalable Queue-based Spin Locks. In: The 21st Annual Symposium on Principles of Distributed Computing (July 2002)
13. Scott, M.L., Scherer III, W.N.: Scalable Queue-based Spin Locks with Timeout. In: The 8th ACM Symposium on Principles and Practice of Parallel Programming (June 2001)
14. Valois, J.D.: Lock-Free Linked Lists Using Compare-and-Swap. In: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, pp. 214–222 (1995)