Deterministic Abortable Mutual Exclusion with Sublogarithmic Adaptive RMR Complexity

Adam Alon  
Blavatnik School of Computer Science  
Tel Aviv University

Adam Morrison  
Blavatnik School of Computer Science  
Tel Aviv University

ABSTRACT

We present a deterministic abortable mutual exclusion algorithm for a cache-coherent (CC) model with read, write, Fetch-And-Add (F&A), and CAS primitives, whose RMR complexity is $O(\log W N)$, where $W$ is the size of the F&A registers. Under the standard assumption of $W = \Theta(\log N)$, our algorithm’s RMR complexity is $O(\log N)$; if $W = \Theta(N^e)$, for $0 < e < 1$ (as is the case in real multiprocessor machines), the RMR complexity is $O(1)$. Our algorithm is adaptive to the number of processes that abort. In particular, if no process aborts during a passage, its RMR cost is $O(1)$.

ACM Reference Format:

1 INTRODUCTION

Mutual exclusion [9] is a fundamental problem in distributed computing. A mutual exclusion object (henceforth, lock) prevents simultaneous entry to a critical section of code, in which some shared resource is accessed. To execute the critical section, a process must first acquire the lock by executing an entry section. The lock algorithm guarantees that at any time, at most one process holds the lock. After acquiring the lock and executing the critical section, a process releases the lock by executing an exit section.

Classic locks do not allow a process waiting in the entry section to abort its lock acquisition attempt and quit the lock protocol. Several use cases, however, require this feature: (1) a process blocked on a lock may wish to abandon its work chunk and switch to working on a different work chunk not subjected to serialization [8]; (2) database systems use aborts to recover from deadlocks and to deal with preemption of a lock holding process [25]; and (3) low-priority processes can abort to expedite lock handoff to a high-priority process [8, 24]. To meet these demands, an abortable lock [24, 25] additionally allows a process to abort its lock acquisition attempt in a finite number of its own steps.

We investigate the remote memory references (RMR) complexity of abortable locks. The RMR complexity measure captures the fact that the cost of a memory reference on shared-memory multiprocessor machines is not uniform. Some references can be satisfied quickly from memory local to the processor, whereas the rest must be satisfied from remote memory. For example, in a cache-coherent (CC) system, each processor stores local copies of the shared variables it accesses in its cache; a cache coherence protocol maintains the consistency of the copies in the different caches. A memory access to a cached variable is local; otherwise, it is remote. In a distributed shared-memory (DSM) system, each shared variable is permanently locally accessible to a single processor and remote to all other processors. References to remote memory are orders of magnitude slower than local memory accesses, as they must traverse the system’s interconnect, and so the performance of lock algorithms critically depends on the number of remote memory references they generate [5, 21]. The RMR complexity measure (or RMR cost) therefore charges an algorithm only for RMRs; local memory accesses are considered free. RMR cost has been used almost exclusively as the complexity measure in shared-memory mutual exclusion research over the last 20 years [4, 6, 7, 10–12, 14, 15, 17, 23].

There is a gap between the known RMR cost of locks and of abortable locks. For an $N$-process system with read, write, comparison primitives [4] such as Compare-And-Swap (CAS), or LL/SC, Yang and Anderson’s lock [26] and Jayanti’s abortable lock [17] both have $O(\log N)$ RMR cost, which is optimal [6]. For mutual exclusion, the $O(\log N)$ lower bound can be defeated by leveraging additional synchronization primitives. The MCS lock [21], for example, uses Fetch-And-Store (SWAP) in addition to the aforementioned primitives, and has RMR cost of $O(1)$. For abortable mutual exclusion, however, no algorithm with worst-case sublogarithmic RMR cost is known. Lee’s abortable lock [19] leverages SWAP and Fetch-And-Add (F&A) primitives to obtain an RMR cost of $O(A^2)$, where $A$ is the number of processes that abort. Lee’s algorithm therefore incurs $O(1)$ RMRs if no process aborts, but its worst-case RMR cost is $O(N^2)$. Sublogarithmic RMR cost can also be achieved using randomization, both for locks [7, 11, 14, 15] and for abortable locks [12, 23], but this paper considers deterministic algorithms.

Our Contribution. We show that, as with mutual exclusion, abortable locks can leverage additional primitives to obtain sublogarithmic worst-case RMR cost. We present an abortable lock algorithm for a CC model with F&A in addition to read, write, and CAS primitives, whose worst-case RMR cost is $O(\log W N)$, where $W$ is the size of the F&A registers. Under the standard assumption of $W = \Theta(\log N)$, this time/space tradeoff implies that our algorithm’s RMR cost is $O(\log N \log N)$. The RMR cost becomes $O(1)$ if $W = \Theta(N^e)$ for some $0 < e < 1$, as is the case in realistic multiprocessor systems. Our algorithm is adaptive to the number of processes that abort. The RMR cost of a complete passage (entry and corresponding exit of the critical section) is $O(\log W A_i)$, where $A_i$ is the number of processes that abort during the passage. If no

---

1E.g., for a system with a billion processes and 64-bit memory words, $W = N^{1/5}$. 

PODC ‘18, July 23–27, 2018, Egham, United Kingdom  
© 2018 Copyright held by the owner/author(s); Publication rights licensed to ACM.  
This is the author’s version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in PODC ‘18: ACM Symposium on Principles of Distributed Computing, July 23–27, 2018, Egham, United Kingdom, https://doi.org/10.1145/3212734.3212759.
process aborts during a passage, its RMR cost is \( O(1) \). The RMR cost of an aborted passage is \( O(\log \alpha \cdot A_i) \), where \( A_i \) is the number of processes that abort during the entire execution. (Note that our notion of adaptivity differs from that of Jayanti’s algorithm [17], in which the RMR cost of a passage depends on the maximum number of processes that are concurrently active during the passage, not only on those that abort.)

We obtain our algorithm by composing two constructions. We first design a one-shot abortable lock, which each process can attempt to acquire at most once. This lock has the aforementioned RMR cost and linear space complexity. We next present a generic transformation that converts a one-shot abortable lock algorithm with space complexity \( s(N) \), where \( s(N) \cdot 2^W = O(1) \), into a long-lived algorithm with the same asymptotic RMR cost and space complexity \( O(N \cdot s(N) + N^2) \). Our final algorithm thus uses \( O(N^2) \) memory words. The transformation does not maintain fairness guarantees. While our one-shot algorithm is first-come-first-served (FCFS) [18], the final algorithm is starvation-free. Table 1 compares the final algorithm to prior work.

### 2 MODEL AND PROBLEM STATEMENT

**Model.** We consider an asynchronous shared-memory model, in which a set of \( N \) deterministic processes communicate by executing atomic operations on shared \( W \)-bit words that support read, write, CAS, and F&A operations. CAS(\( w, o, n \)) atomically changes \( w \)'s value to \( o \) if \( w \) contains \( o \), and returns true; otherwise, it returns false without modifying \( w \). F&A(w, x) atomically updates the value stored in \( w \) from \( v \) to \( v + x \), and returns \( v \). A configuration consists of the state of all processes and memory words. An execution is a (possibly infinite) sequence of steps. Each step consists of a process invoking an operation on a shared variable and receiving its return value, thereby moving the system to a new configuration.

**RMR complexity.** In a cache-coherent (CC) model, each processor maintains local copies of shared variables it accesses in its cache, and a coherence protocol ensures the consistency of cached variables by invalidating cached copies when a variable is written. A memory access to an uncached variable is called a remote memory reference (RMR). Each write, CAS, or F&A incurs an RMR. A read by process \( p \) of a shared variable \( w \) incurs an RMR if (1) it is \( p \)'s first read of \( w \), or (2) after \( p \)'s last read of \( w \), another process performed a write, CAS, or F&A to \( w \). In a distributed shared-memory (DSM) model, each register is (forever) local to some processor and remote to all others. An access to a remote register is an RMR.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Model</th>
<th>Primitives</th>
<th>Space (#words)</th>
<th>Fairness Guarantee</th>
<th>Worst-case RMR cost of a passage through CS</th>
<th>Adaptive bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scott [24]</td>
<td>CC/DSM</td>
<td>SWAP, CAS</td>
<td>unbounded</td>
<td>FCFS</td>
<td>unbounded</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Jayanti [17]</td>
<td>CC/DSM</td>
<td>LL/SC or CAS</td>
<td>( O(N) )</td>
<td>FCFS</td>
<td>( O(\log N) )</td>
<td>( O(\text{min}(k, \log N)) )</td>
</tr>
<tr>
<td>Lee [27]</td>
<td>CC</td>
<td>F&amp;A, SWAP</td>
<td>( O(N^2) )</td>
<td>FCFS</td>
<td>( O(N^2) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>This Work</td>
<td>One-Shot</td>
<td>CC/DSM</td>
<td>F&amp;A</td>
<td>FCFS</td>
<td>( O(\log W \cdot N) )</td>
<td>( O(\log W \cdot A_i) ) for complete passage</td>
</tr>
<tr>
<td></td>
<td>Long-Lived</td>
<td>CC</td>
<td>F&amp;A, CAS</td>
<td>( O(N^2) )</td>
<td>Freedom</td>
<td>( O(\log W \cdot A_i) ) for aborted passage</td>
</tr>
</tbody>
</table>

Table 1: Comparison of abortable locks, showing their system model, required primitives, complexity, and fairness.

**Abortable mutual exclusion.** An abortable lock supports the methods Enter and Exit. A process attempts to acquire the lock by executing Enter. If Enter returns true, the process acquires the lock and enters the critical section (CS). While inside Enter, a process can receive an external signal to abort its attempt, in which case Enter may return false. Once a process completes the CS, it exits the CS by invoking Exit and thereby releases the lock. A passage is the sequence of steps in which a process executes Enter, the CS, and Exit. A process not executing Enter, Exit, or the CS is said to be in the remainder section.

**Problem statement.** The goal is to design an abortable lock algorithm satisfying the following requirements: (1) mutual exclusion: at any time, at most one process is in the CS; (2) starvation-freedom: if no process crashes outside the remainder section and every process that enters the CS eventually leaves it, then if a process \( p \) invokes Enter and does not abort its attempt, \( p \) eventually enters the CS in that attempt; (3) bounded exit: a process completes an Exit call in a finite number of its own steps; and (4) bounded abort: if a process \( p \) busy waiting in Enter receives a signal to abort its attempt, then \( p \)'s execution of Enter returns (either true or false) within a finite number of \( p \)'s steps.

### 3 ONE-SHOT ALGORITHM

Here, we describe our one-shot abortable lock, which is the main building block of our algorithm. Unless noted otherwise, we assume the CC complexity model.

The one-shot lock implements an array-based queue lock [5, 13], augmented with a data structure that tracks which processes have aborted and thus given up their place in the queue. Our high-level design is similar to Jayanti’s algorithm [17], with the main difference being in the augmenting data structure. Whereas Jayanti uses a linearizable priority queue [16], we use a tree-based ordered set that is not linearizable. The semantics of our Tree data structure cannot be cleanly captured by a sequential specification, as they depend on the concurrency between operations.

Figure 1 shows the pseudo-code of the lock algorithm. Its underlying data structures are an array-based queue, go, and a Tree whose implementation we describe in Section 4. We assume that each process attempts to perform at most one pass through the critical section. To acquire the lock, a process first increments Tail using F&A to obtain an index to a slot in the go array. We will identify each process with its index \( 0 \leq i \leq N - 1 \). Process \( i \) spins

2Returning false is not mandatory because a process can receive the signal after being handed the lock, but before noticing the handoff. Formulations in which an abort signal moves the process to some Abort method [17] similarly require this method to detect and handle such a scenario.
Algorithm 3.1 Enter()

1: \( i \leftarrow F\&A(Ta1, 1) \)
2: while \(-go[i] \) do
3: \( \text{if } \text{AbortSignal then} \)
4: \( \text{Abort()} \)
5: return false
6: \( \text{Head} \leftarrow i \)
7: return true

Algorithm 3.2 Exit()

8: \( \text{head} \leftarrow \text{Head} \)
9: \( \text{LastExited} \leftarrow \text{head} \)
10: \( \text{SignalNext}(\text{head}) \)

Algorithm 3.3 Abort(i)

11: \( \text{Tree. Remove}(i) \)
12: \( \text{head} \leftarrow \text{Head} \)
13: \( \text{if } \text{head} \neq \text{LastExited then} \)
14: \( \text{return} \)
15: \( \text{SignalNext}(\text{head}) \)

Algorithm 3.4 SignalNext(head)

16: \( j \leftarrow \text{Tree. FindNext}(\text{head}) \)
17: \( \text{if } j \in \{ \top, \bot \} \) then
18: \( \text{return} \)
19: \( \text{go}[j] \leftarrow \text{true} \)

Figure 1: One-shot abortable lock algorithm

on its slot until being signaled that it owns the lock, and then sets
the queue’s Head to i and enters the CS. (Initially, go[0] is set, so
process 0 immediately enter the CS.)

When process i exits the CS, it hands the lock off to the next slot
in the queue, which is the minimum j > i such that slot j has not been
abandoned by its process due to an abort. The Tree data structure
facilitates this handoff. It maintains the (ordered) set of queue slots
that have not been abandoned by aborting processes. (Initially,
Tree = \{0, . . . , N - 1\}.) The handoff of process i’s lock ownership
is performed by SignalNext(i). This procedure calls Tree.FindNext(i)
to find the next slot, j, in order to set go[j] to true. FindNext(i) might return \( \bot \), indicating that all possible successor slots
have not been abandoned and thus the lock is now unusable. Finally,
FindNext(i) may also return \( \top \), indicating that its successor
search “crossed paths” with an aborting process k removing itself
from Tree. In such a case, as described next, aborting processes will
assume responsibility for performing the handoff on behalf of
process i.

Aborts are performed by the Abort() procedure, which a process
busy waiting in Enter calls when it detects the external AbortSignal.
An aborting process i abandons its queue slot by removing itself
from Tree. It then reads Head to obtain the id of the process in the
CS, h, and compares h to the LastExited variable, which contains
the id of the last process to release the lock. If h = LastExited,
then process h may be in the middle of exiting the CS, and its
FindNext() might have crossed paths with process i’s Remove and
thus returned \( \top \). Therefore, process i assumes responsibility for h’s
lock handoff and executes SignalNext(h). Of course, it can then
cross paths with some aborting process j and so fail to complete
the handoff, but then process j would assume responsibility for
the handoff. The crux of our correctness proofs is to show that
eventually, some aborting process that assumes responsibility for
the handoff manages to complete it.

A DSM variant. In the DSM model, we cannot guarantee that the
go slot of a process is local, since the slot is determined at run time.
As a result, a process might incur an unbounded number of RMRs
while busy waiting. We use indirection to address this problem, by
having the process spin on a local spin bit that it publishes in an
announce array. To synchronize with a concurrent lock handoff,
process i publishes its spin bit s by writing announce[i] = s, and
then, if go[i] \neq 1, spinning locally on s. A SignalNext() hands the
lock to process i by writing go[i] = 1, reading s = announce[i], and
if s \neq \bot, writing s = 1.

4 TREE DATA STRUCTURE
The Tree data structure maintains a W-ary tree with N leaves, whose
height is \( H = \lceil \log_W N \rceil \). A tree node u contains a W-bit word,
initially 0, in which the j-th most significant bit is associated with
u’s j-th child, counting from the left (so the leftmost bit in u is
associated with u’s leftmost child). We number the leaves from left
to right starting with 0, and identify leaf p with process p (equivalently,
queue slot p). Tree has the property that if some queue slot in the
subtree rooted at u has not been abandoned by an aborting process,
then the bit associated with u in u’s parent is clear. To maintain
this property, an aborting process p ascends the tree starting from
its leaf, performing a F&A to set the bit associated with its subtree
in each visited node u. If all bits in u are then set, the process keeps
ascending to u’s parent; otherwise, it stops.

FindNext(p) needs to find the first leaf to the right of p that
has not been abandoned. To find this leaf, it simply walks up the
tree until finding a clear bit to the right of p, and then walks down
towards the relevant leaf. If FindNext(p) does not find a zero bit
during its ascent, it returns \( \bot \). If it encounters a node in which all
bits are set after starting to descend, then it has “crossed paths” with
a Remove() ascending the subtree and so returns \( \top \). Figure 2
depicts these scenarios.

Figure 3 presents the pseudo-code of the algorithm. Because the
tree structure is static, we do not need pointers in the nodes; parent
child nodes are computed by the processes. Leaf nodes act as
static sentinels. Only the values stored in non-leaf nodes need to be
stored in shared memory. The size of the tree is thus \( O(N^{\frac{W}{2}}) \) words.

For \( 0 \leq j \leq W - 1 \), we denote the j-th child (from left to right)
of node u by Child(u, j); if u is a leaf, Child(u, j) = \( \bot \). We denote
parent of node u by Parent(u); if u is the root, Parent(u) = \( \bot \). The
i-th leaf is denoted Leaf(i). The W bits maintained in a node are
stored in a field named value. For leaves, value contains the id of
the associated process (i.e., Leaf(p).value = p for \( 0 \leq p \leq N - 1 \)).

We use the following notation to associate bits with processes (equivalently, queue slots). The level of node u, denoted Lvl(u), is 0
if u is a leaf; otherwise, Lvl(u) = Lvl(Child(u, 0)) + 1. Let \( 1 \leq l \leq H \).
The node of process p in level l, denoted Node(p, l), is Leaf(p) if
l = 0; otherwise, Node(p, l) = Parent(Node(p, l - 1)). The offset
process p in level l, denoted Offset(p, l) is the number o such that
Child(Node(p, l), o) = Node(p, l - 1). The bit of process p in level
l, denoted Bit(p, l), is the o-th MSB of Node(p, l).value, where
o = Offset(p, l).

Figure 2: Possible FindNext(p) scenarios
Algorithm 4.1 Tree.FindNext(p)

20: for i = 1 to H do
21: node = Node(p, i)
22: offset = Offset(p, i)
23: snap = node.value
24: if HasZeroToTheRight(snap, offset) then
25: break
26: if HasZeroToTheRight(snap, offset) = false then
27: return ⊥ // reached root and found no candidate
28: index ← GetFirstZeroToTheRight(snap, offset)
29: node ← Child(node, index)
30: for dummy ← 1 to 1 do
31: snap = node.value
32: if snap = EMPTY then
33: return ⊥
34: index ← GetFirstZero(snap)
35: node ← Child(node, index)
36: return node.value

Algorithm 4.2 Tree.Remove(p)

37: for i = 1 to H do
38: j ← word with only Offset(p, i)-th MSB set
39: snap = Fetch(Node(p, i), value, j)
40: if snap + j ≠ EMPTY then
41: break

Algorithm 4.3 Tree.AdaptiveFindNext(p)

42: node ← Node(p, 1)
43: offset ← Offset(p, 1)
44: for i = 1 to H do
45: if offset = W − 1 then
46: node ← RightCousin(node)
47: offset ← −1
48: snap = node.value
49: if HasZeroToTheRight(snap, offset) then
50: break
51: if offset = −1 then
52: offset ← OffsetAtParent(node) − 1
53: else
54: offset ← OffsetAtParent(node)
55: node ← Parent(node)
56: Continue as in FindNext(f) (from Line 26 of Algorithm 4.1)

4.1 Adaptive FindNext()

It is easy to see that the RMR cost of Remove() is \( O(\log W \cdot A_t) \), where \( A_t \) is the number of processes that abort during the execution. However, FindNext() is not adaptive to the number of aborts. When invoked on a leaf \( v \) that is the rightmost node in its subtree, FindNext() ascends to the root of the subtree—which can be of height \( \log W \cdot N \)—even if the leaf of the next non-aborted process is immediately to the right of \( v \) (just in another subtree).

We introduce a novel (though small) change to the way FindNext() walks up the tree that improves its RMR cost to \( O(\log W \cdot A_t(p)) \), where \( A_t(p) \) is the number of processes that abort during process \( p \)'s passage. Algorithm 4.3 presents the pseudo-code of the adaptive algorithm. Instead of ascending along a path from the leaf to the root, whenever we reach a node \( u \) that is the rightmost child of its parent, we “sidestep” to \( v \), the node right to \( u \)’s parent at the same level, as depicted in Figure 4. The idea is that because we cannot hope to find a zero bit to the right of \( u \)'s bit in Parent\((u)\), we instead optimistically check whether \( v \) has a zero bit. If this is the case, then FindNext\((p)\)'s ascent would have stopped at \( w \), the lowest common ancestor of \( u \) and \( v \), and would have then started descending until eventually reaching \( v \). If, however, \( v \) does not contain a zero bit, then all leaves between \( p \) and the rightmost leaf in \( v \)'s subtree have been abandoned. Therefore, it is safe to resume the ascent from \( v \).

A subtle point in Algorithm 4.3 is that if we “sidestep” to \( v \), fail to find a zero bit there, and ascend to \( v \)'s parent, we still need the search for a zero bit in \( v \)'s parent to include \( v \)'s subtree. (Hence the use of OffsetAtParent\((v)\) = 1 in line 52, after “sidestepping.”) The reason is that the Remove() which has set the last bit in \( v \) might not have set \( v \)'s bit in the parent yet. In such a case, FindNext() would have returned \( T \), as it would descend towards \( v \) (which it would find EMPTY) after going through Parent\((v)\) (where \( v \)'s bit is zero). Therefore, AdaptiveFindNext() mimics this property.

We can thus show that AdaptiveFindNext() is equivalent to FindNext() in the following sense: (The proof appears in the full version [3].)

Lemma 1. Let \( E \) be an execution of the one-shot algorithm using AdaptiveFindNext(). Then there exists an execution \( E' \) of the one-shot algorithm using FindNext(), such that (1) process \( p \) invokes FindNext\((q)\) in \( E' \) if and only if it invokes AdaptiveFindNext\((q)\) in \( E \); (2) process \( p \)'s FindNext\((q)\) in \( E' \) returns \( x \in [N] \cup \{⊥, \top\} \) if and only if its AdaptiveFindNext\((q)\) in \( E \) returns \( x \); and (3) the order of FindNext() invocation/responses in \( E' \) is the same as the order of AdaptiveFindNext() invocation/responses in \( E \).

5 CORRECTNESS AND COMPLEXITY OF THE ONE-SHOT ALGORITHM

Here, we prove that our one-shot algorithm satisfies mutual exclusion, starvation freedom, and the first-come-first-served (FCFS) fairness condition (formally defined later). Then we show that the algorithm has sublogarithmic adaptive RMR cost. We assume that \( W \geq \lceil \log N \rceil \). The following theorem summarizes these properties.

Theorem 2. The one-shot lock of Figure 1 and Figure 3 satisfies mutual exclusion, starvation freedom, bounded exit, bounded abort, and FCFS. Each passage of the algorithm incurs \( O(\log W \cdot A_t) \) RMRs, where \( A_t \) is the number of processes that abort during the passage. An aborted attempt incurs \( O(\log W \cdot A_t) \) RMRs, where \( A_t \) is the number of processes that abort during the entire execution.

Lemma 1 implies that any correctness result obtained for the one-shot algorithm using FindNext() (Algorithm 4.1) also holds.

Figure 3: Tree data structure

Figure 4: Comparison of FindNext() ascent algorithms
for AdaptiveFindNext() (Algorithm 4.3). Therefore, for simplicity, the following proofs consider the one-shot lock with FindNext().

**Notation.** Let $E$ be an execution. We denote by $E_t$ the prefix of $E$ consisting of the first $t$ steps in $E$. The value of variable $v$ at time $t$ is the value of $v$ in the configuration following $E_t$. Step $s$ occurs at time $t$ if it is the $t$-th step in $E$. We identify a process with the index it obtains from the F&A on Head (Algorithm 3.1, line 1).

### 5.1 Tree Properties

We begin the proof of Theorem 2 by proving some properties of the Tree data structure. We assume a well-formed execution, in which (1) each process invokes FindNext() at most once; (2) Remove(p) is invoked only by process $p$, and at most once; and (3) if process $p$ invokes both Remove(p) and FindNext(u), then it invokes them in this order. An execution of the one-shot algorithm is well-formed.

#### 5.1.1 Preliminaries.

Abusing notation, we say that $p \in u$ if Node(p, lув(u)) = $u$, i.e., if Leaf(p) is in the subtree rooted at $u$.

**Definition 1.** The lowest common level of leaves $p$ and $q$, denoted LCL($p, q$), is the lowest level in which $p$ and $q$ have a common ancestor, i.e., LCL($p, q$) = min{l | $p \in \text{Node}(p, l)$ and $q \in \text{Node}(q, l)$}. The lowest common level of leaf $p$ and internal node $u$ is LCL($p, u$) = LCL($p, q$) for some leaf $q \in u$.

The structure of the tree immediately implies that for any leaves $p < q$ with $L = \text{LCL}(p, q)$, the following hold: (1) Offset(set(p, L)) < Offset(set(q, L)); (2) $\forall l \geq L$. Offset(set(p, l)) = Offset(set(q, l)); (3) $\forall l < L$. Node(p, l) = Node(q, l); and (4) $\forall p < r < q$. LCL($p, r$) = LCL($r, q$) ≤ L.

**Claim 3.** In any execution $E$, Bit(p, lув) = 1 if and only if some process performs F&A(Node(p, lув), value, j) in Remove() (Algorithm 4.2, line 39), where $j$ is a word whose only set bit is the Offset(set(p, lув))-th MSB.

**Proof.** Appears in the full version of the paper [3].

**Lemma 4.** For every execution $E$, process $p$, and $1 \leq lув < H$, if Bit(p, lув) = 1 at time $t$, then for all $l < lув$ and $m \in \text{Node}(p, lув)$, Bit(m, l) = 1 at time $t$.

**Proof.** We prove by induction over lув. The base case of $lув = 1$ is vacuously true. For the inductive step, assume the claim holds for levels 1, ..., lув-1. Suppose that Bit(p, lув) = 1 at time $t$. Claim 3 implies that Bit(p, lув) = 1 because of a F&A performed on Node(p, lув),value. Then Node(q, lув) = Node(p, lув) and Offset(set(q, lув)) = Offset(set(p, lув)). Therefore, $q \in \text{Node}(p, lув)$, and for process $q$ to have accessed Node(p, lув) to set Bit(p, lув), it must have reached iteration lув in Remove(q). Consider iteration lув - 1 of Remove(q). Process $q$ wrote Node(q, lув - 1), value = EMPTY at time $t_1 < t$. Because Node(q, lув - 1) = Node(p, lув - 1), this means that for every $m \in \text{Node}(p, lув)$, Bit(m, lув - 1) = 1 at time $t_1$. Let $m \in \text{Node}(p, lув)$. We know Bit(m, lув - 1) = 1 at time $t_1$. Since $m \in \text{Node}(m, lув - 2)$, by the induction assumption, for every $l < lув - 1$, Bit(m, l) = 1 at time $t_1$. The claim follows, as $t_1 < t$.

**Corollary 5.** Remove Invariant. For every execution $E$, process $p$, and $1 \leq lERVED < H$, if Bit(p, lERVED) = 1 at time $t$, then (1) for every $q \in \text{Node}(p, lERVED - 1)$, process $q$ invokes Remove(q) at some time $t'$ ≤ $t$; and (2) if Remove(p) is not invoked in $E_t$, then for all $1 \leq lERVED < H$, Bit(p, lERVED) = 0 in $E_t$.

#### 5.1.2 FindNext() Properties

Let $E$ be an execution of the algorithm. We now prove several properties provided by Tree during $E$. We identify the invocation (respectively, completion) of a Tree method with the first (respectively, last) memory operation performed by the method’s code. If process $p$ invokes method $M$, we denote by $M_p$ the subsequence of $E$ that starts at $M$’s invocation and ends at its completion (or, if $M$ does not complete, at the end of $E$). We say that $M_p$ happens before $M'_q$, denoted $M_p \rightarrow M'_q$, if the completion of $M_p$ occurs before the invocation of $M'_q$ (i.e., $p$ performs the last memory operation in $M_p$’s code before $q$ performs the first memory operation in $M'_q$). We say that a method $M_p$ starts before method $M'_q$ completes, denoted $M_p \prec M'_q$, if the invocation of $M'_q$ occurs before the completion of $M_p$ in $E$. Note that $M_p \prec M'_q$ is equivalent to $M'_q \prec M_p$. We use the fact implied by this equivalence, that if $A \prec B$ and $B \prec C$, then $A \prec C$.

The first three properties say that FindNext(p) returns the first process $q > p$ that did not yet invoke Remove(v).

**Property 6.** If FindNext(p) returns $q \not\in \{\bot, \top\}$, then $p < q$.

**Property 7.** If FindNext(p) returns $q \not\in \{\bot, \top\}$ and Remove(q) is invoked in $E$, then FindNext(p) → Remove(q).

**Proof.** Let $t_1$ be the time a executes the last step of FindNext(p). Code inspection shows that this step reads Bit(q, 1) = 0. The first step of Remove(q) is a F&A that sets Bit(q, 1) to 1. It follows that Remove(q) cannot perform its first step before $t_1$, i.e., FindNext(p) → Remove(q).

**Corollary 8.** If Remove(q) → FindNext(p) then FindNext(p) does not return $q$.

**Property 9.** If FindNext(p) returns $q \not\in \{\bot, \top\}$, then for all $p < w < q$, Remove(w) → FindNext(p).

**Proof.** Let $t$ be the time a executes the last step of FindNext(p). Assume towards a contradiction, that for some $p < w < q$, Remove(w) does not perform its first memory operation in $E_t$. It follows from the remove invariant that for all $1 \leq lERVED < H$, Bit(w, lERVED) = 0 at time $t$. Let $l_p, q = LCL(p, q), l_p, w = LCL(p, w)$, and $l_{w, q} = LCL(w, q)$. Code inspection shows that a ascends until level $l_p, q$, observing only 1 bits to the right of Offset(set(L)) at all levels $l < l_p, q$. At level $l_p, q$, a observes 0 at bit Offset(set(q, l_p, q)) and 1 at every bit between Offset(set(p, l_p, q)) and Offset(set(q, l_p, q)). Then a starts descending. In each level $l$, a observes 0 at bit Offset(set(q, l)) and 1 at every bit to the left of that offset. Finally, a reaches Leaf(q) and returns $q$.

If $l_p, w < l_p, q$, then Offset(set(w, l_p, w)) > Offset(set(p, l_p, w)). Yet a ascends through level $l_p, w$, implying that it observes Bit(w, l_p, w) = 1 during $E_t$, a contradiction. If $l_p, q = l_p, q = l_{w, q}$, then Offset(set(p, l_p, w)) < Offset(set(q, l_p, w)). Thus, a reads Bit(w, l) = 1 during $E_t$, a contradiction. Otherwise, $l_p, q > l_{w, q} > 0$, and therefore Offset(set(w, l_w, q)) < Offset(set(q, l_w, q)). Yet
a descends level $I_{w,q}$, implying that it observes $\text{Bit}(w, I_{w,q}) = 1$ during $E_l$, a contradiction. □

The next property says that if $\text{FindNext}(p)$ returns $\perp$, then every process $q > p$ has invoked $\text{Remove}(q)$ (and thus no next process should be signaled).

**Property 10.** If $\text{FindNext}(p)_a$ returns $\perp$, then for all $p < q < N$, $\text{Remove}(q) \rightsquigarrow \text{FindNext}(p)_a$.

**Proof.** Assume towards a contradiction, that for some $p < w < n$, $\text{Remove}(w)$ does not perform its first step in $E_l$. It follows from the remove invariant that for all $1 \leq l \leq H$, $\text{Bit}(w, l I_l) = 0$ at time $t$. Let $I_{p,w} = \text{LCI}(p, w)$ ≤ $H$. At level $I_{p,w}$, $a$ observes 1 at all offsets greater than $\text{OffSet}(p, I_{p,w})$ at some time $t_1 \leq t$. We know $\text{OffSet}(p, I_{p,w}) < \text{OffSet}(w, l I_l)$, implying a observed $\text{Bit}(w, I_{p,w}) = 1$ at time $t_1 \leq t$, which is a contradiction. □

The next property says that the process ids returned by non-overlapping $\text{FindNext}(p)$ executions are monotonically increasing.

**Property 11.** If $\text{FindNext}(p)_a \rightarrow \text{FindNext}(p)_b$ and $\text{FindNext}(p)_b, \text{FindNext}(p)_a$, respectively return $q_a, q_b$, then $q_a \leq q_b$.

**Proof.** Assume towards a contradiction that $q_a > q_b$. From code inspection, a ascends until level $I_{p, q_a} = \text{LCI}(p, q_a)$ and then descends until level 0, and $b$ ascends until level $I_{p, q_b} = \text{LCI}(p, q_b)$ and then descends until level 0. From Property 6, we have $p < q_b < q_a$, and therefore $I_{p, q_b} < I_{p, q_a}$. If $I_{p, q_b} < I_{p, q_a}$, then (1) $a$ reads $\text{Bit}(q_b, I_{p, q_b}) = 1$, as a ascends past level $I_{p, q_b}$; and (2) $b$ reads $\text{Bit}(q_b, I_{p, q_b}) = 0$, as it stops its ascent at level $I_{p, q_b}$. However, $\text{FindNext}(p)_a \rightarrow \text{FindNext}(p)_b$, and therefore any bit observed by $a$ as 1 cannot be observed by $b$ as 0, so this is a contradiction.

If $I_{p, q_b} = I_{p, q_a}$, then both $a$ and $b$ stop their ascent at this level, but descend into different subtrees. Let $l = \text{LCI}(q_a, q_b)$. If $I_{p, q_b} = I_{p, q_a}$, then clearly $\text{OffSet}(l, I_l) < \text{OffSet}(q_b, I_l) < \text{OffSet}(q_a, l)$, and therefore $a$ reads $\text{Bit}(q_b, I_l) = 1$ but $b$ reads $\text{Bit}(q_b, l) = 0$, which is a contradiction, since $\text{FindNext}(p)_a \rightarrow \text{FindNext}(p)_b$. Therefore, $l < I_{p, q_b} = I_{p, q_a}$, and $\text{OffSet}(q_b, l) < \text{OffSet}(q_a, l)$. Thus, $a$ reads $\text{Bit}(q_b, l) = 1$, as it descends towards $a$, but $b$ reads $\text{Bit}(q_b, l) = 0$. Since $\text{FindNext}(p)_a \rightarrow \text{FindNext}(p)_b$, this is a contradiction. □

The last property refers to scenarios in which process $a$’s $\text{FindNext}(p)$ is about to return some value $q > p$, but a $\text{Remove}(q)$ crosses paths with $a$’s $\text{FindNext}(p)$ execution and causes it to return $\perp$. The property says that in such a case, there exists a process $b$ that assumes responsibility for $p$’s lock handoff. We reason about the responsibility for the handoff with the following responsibility relation.

**Definition 2** (Responsibility relation). Given processes $a, b$, we say that process $a$ has less $p$-responsibility than $b$, denoted $a <^R_p b$, if the following 4 conditions hold in $E_l$:

1. $\text{Remove}(b)$ is invoked by $b$ in $E_l$.
2. $\text{FindNext}(p)_a$ is invoked by $a$ in $E_l$.
3. $\text{FindNext}(p)_a \rightarrow \text{Remove}(b)$.
4. For every $p < d < \max\{a, b\}$, $\text{Remove}(d) \rightsquigarrow \text{Remove}(b)$.

**Lemma 12.** The relation $<^R_p$ is a strict partial order.

**Proof.** A strict partial order is irreflexive and transitive. Irreflexive: Assume $\text{Remove}(a)$ and $\text{FindNext}(p)_a$ are both invoked in $E$ by some process $a$. The execution is well-formed, and so we have $\text{Remove}(a) \rightarrow \text{FindNext}(p)_a$. Therefore, $\neg(\text{FindNext}(p)_a \rightarrow \text{Remove}(a))$. Therefore, Conditions 1–3 in Definition 2 cannot all hold together for $a$, and thus $a <^R_p a$ cannot hold.

Transitive: Assume $a <^R_p b$ and $b <^R_p c$ for some processes $a$, $b$, and $c$. From Conditions 1 and 2 in Definition 2, $\text{Remove}(c)$ and $\text{FindNext}(p)_a$ are invoked in $E$. Therefore, Conditions 1 and 2 also hold for $a$ and $c$. From Condition 1, $\text{Remove}(b)$ is invoked by $b$ in $E$. From Condition 2, $\text{FindNext}(p)_b$ is invoked by $b$ in $E$. The execution is well-formed, so we have $\text{Remove}(b) \rightarrow \text{FindNext}(p)_b$. From Condition 3, $\text{FindNext}(p)_a \rightarrow \text{Remove}(c)$. Therefore, we conclude that $\text{FindNext}(p)_a \rightarrow \text{Remove}(b) \rightarrow \text{FindNext}(p)_b \rightarrow \text{Remove}(c)$, and thus $\text{FindNext}(p)_a \rightarrow \text{Remove}(c)$ holds. Therefore Condition 3 holds for $a, c$. Let $d$ such that $p < d < \max\{a, c\}$. If $p < d < \max\{b, c\}$, then from Condition 4 we get $\text{Remove}(d) \rightarrow \text{Remove}(c)$ as needed. Otherwise, $p < d < \max\{a, b\}$, and from Condition 4, we know $\text{Remove}(d) \rightarrow \text{Remove}(b)$. We can thus conclude that $\text{Remove}(d) \rightarrow \text{Remove}(b) \rightarrow \text{FindNext}(p)_b \rightarrow \text{Remove}(c)$, which implies $\text{Remove}(d) \rightarrow \text{Remove}(c)$ as needed. Therefore, Condition 4 holds for $a, c$, implying $a <^R_p c$. □

**Corollary 13.** If $a <^R_p b$, then there exists a maximal $u$ for $a$, i.e., $u$ such that $a <^R_p u$ and for all $c$, $\neg(u <^R_p c)$.

**Property 14.** Fixes processes $p$ and $a \geq p$. If for every $p < d < a$, $\text{Remove}(d) \rightarrow \text{Remove}(a)$, and $\text{FindNext}(p)_a$ returns $\perp$, then for some process $b \neq a, a <^R_p b$.

**Proof.** Since $\text{FindNext}(p)_a$ returns $\perp$, $a$ reads $\text{EMPTY}$ from node.value for some node node at time $t$ (Algorithm 4.1, line 31). Let $b$ be the process whose F&A in $\text{Remove}(b)$ writes $\text{EMPTY}$ to node.value. Clearly, $b \neq a$ and $b \neq a$. We claim that $a <^R_p b$.

*Condition 1.* Let $l^G$ be the level of node, i.e., the level in which $p$ stops its descent. Then at time $t_1$, $a$ reads $\text{Bit}(l^G + 1, b) = 0$, and at time $t_2 > t_1$, $a$ reads $\text{Bit}(l^G, b) = 1$. The latter fact implies, from the remove invariant, that $\text{Remove}(b)$ is invoked by $b$ in $E$.

*Condition 2.* Immediate from the premise.

*Condition 3.* Consider $\text{Remove}(b)$’s execution. Since $b$ sets node.value to $\text{EMPTY}$, it ascends to level $l^G + 1$, with the intent to set $\text{Bit}(l^G + 1, b)$ to 1. However, at time $t_1$, $a$ reads $\text{Bit}(l^G + 1, b) = 0$, implying that $\text{Remove}(b)$ did not perform a F&A at level $l^G + 1$ by then. Therefore, $\text{Remove}(b)$ does not complete before $\text{FindNext}(p)_a$ is invoked, i.e., $\text{FindNext}(p)_a \rightarrow \text{Remove}(b)$.

*Condition 4.* Let $p < d < \max\{a, b\}$. Suppose $p < d < a$. We have $\text{Remove}(d) \rightarrow \text{Remove}(a), \text{Remove}(a) \rightarrow \text{FindNext}(p)_a$ and $\text{FindNext}(p)_a \rightarrow \text{Remove}(b)$. Therefore, $\text{Remove}(d) \rightarrow \text{Remove}(b)$, as needed. Otherwise, $p \leq d < b$. Let $l^* = b$ be the level at which $a$ stops its ascent and starts descending towards $b$. Let $l^- = \text{LCI}(l^d, b)$, and $l'^- = \text{LCI}(l^d, b)$. Suppose that $l^- < l^*$. If $l'^- < l^*$, $a$ descends through $\text{Nodel}(l^d, l'^-) = \text{Nodel}(l^d, l'^-) < l^*$ and observes $\text{Bit}(d, l'^-) = 1$ since it descends towards $b$. If $l'^- = l^*$, $a$ descends through
For every execution $E$ and process $i$, if $go(i) = true$ at time $t_0$ then for all $j < i$, $j$ performs the first step of either $Abort(i)$ or $Exit()$ at time $t < t_0$.

**Proof.** We prove by contradiction. Let $E$ be an execution in which the claim is false, and let $t_0$ be the first time in which the above condition is violated. Then at time $t_0$, some process $k$ writes $go[i] = true$, but there exists some process $j < i$ that has not yet performed the first step of either $Abort(i)$ or $Exit()$.

We claim that $LastExited < j$ at all times $t < t_0$. Otherwise, at some time $t < t_0$, some process $k' \geq j$ writes $LastExited = k'$, implying that $k'$ exits the CS. By definition of $j$, $k' \neq j$. Process $k'$ reads $go[k'] = true$ at time $t' < t < t_0$, even though $j < k'$ and $j$ has not performed the first step of either $Abort(i)$ or $Exit()$ at $t'$. This contradicts $t_0$ being the first time in which such a violation occurs.

Now, since $k$ writes $go[i] = true$ at time $t_0$, it completes a $FindNext(h)$ invocation, for some $h$, at time $t_1 < t_0$ and receives $i$ as the response. Clearly, $k \neq j$, because by time $t_0$, process $k$ has invoked $FindNext(i)$ but $j$ has not. Process $k$ invokes $SignalNext(h)$ either from $Abort(i)$ or from $Exit()$. In either case, for $SignalNext(h)$ to be invoked, it must hold that at some time $t_2 < t_1$, process $k$ reads $LastExited$ and observes its value to be $h$ (either because of line 13 in Algorithm 3.3 or line 9 of Algorithm 3.2). Thus, $h < j$. Now, $FindNext(h)_k$ returns $i$ at time $t_1 < t_0$. Since $h < j < i$, it follows from Property 9 that $Remove(j) \rightarrow FindNext(h)_k$, i.e., that process $j$ performs the first step of $Remove(j)$ at some time $t' < t_1 < t_0$, which is a contradiction.

**Corollary 16.** The one-shot algorithm satisfies mutual exclusion.

**5.3 FCFS and Starvation Freedom**

For a one-shot abortable lock, the first-come-first-served (FCFS) fairness condition [17] requires that (1) $Enter$ starts with a doorway, which is a wait-free section of code (i.e., that can be completed in a finite number of the executing processes’ steps); and (2) if process $p$ completes the doorway before $q$ starts executing the doorway, and if $p$ does not abort, then $p$ enters the CS before $q$ does.

**Lemma 17.** The one-shot algorithm satisfies FCFS.

**Proof.** We define the doorway to be the F&A operation (Algorithm 3.1, line 1). Process $j$ receives index $j$ from this F&A and proceeds to the "waiting" section, in which it waits for $go[j]$ to become $true$. Let $i$ be a process that executes the doorway after process $j$, receiving index $i > j$. Suppose that at time $t$, process $i$ observes $go[i] = true$ and enters the CS. Lemma 15 implies that by time $t$, process $j$ has invoked either $Exit()$ or $Abort()$, which establishes FCFS.

**Lemma 18.** The one-shot algorithm satisfies starvation freedom.

**Proof.** By contradiction. Let $E$ be an execution in which no process crashes outside the remainder section and every process that enters the CS eventually leaves it, but there exists a minimal $i$ such that process $i$ invokes but does not complete the execution of $Enter()$. This means that $i$ does not invoke $Abort()$, $Exit()$, or $SignalNext()$. It is easy to verify that $LastExited$ and $Head$ are both strictly increasing. Let $m$ be the largest value of $Head$ in $E$. Then $m < i$. Otherwise, some process $m > i$ writes to $Head$, and the assumption on $E$ implies that $m$ eventually enters the CS, which violates FCFS. It is easy to verify that at any time $LastExited \leq Head$ and so $LastExited \leq m$ in $E$. The assumption on $E$ implies that process $m$ eventually, at some time $t_0$, reaches line 9 and sets $LastExited = m$. Therefore, from time to onwards, $LastExited = Head = m$. Let $k$ be the maximal value such that $go[k] = true$ in $E$. Then $m \leq k < i$.

**Claim 19.** There exists $q$ such that $k < q$.

**Proof.** Suppose that $k = m$. Then process $k$ executes $Exit()$, invoking $FindNext(v)_k$, which returns $b$. If $b \notin \{\top, \perp\}$, then from Property 6, $b > k$ and $k$ eventually writes $go[b] = true$, contradicting $k$’s maximality. If $b = \perp$, then Property 10 implies that $i$ invokes $Remove(i)$, which can only be invoked from $Abort(i)$, which is a contradiction. Therefore, $b = \top$. The preconditions for Property 14 hold vacuously, as there is no $d$ such that $m = k < d < k$. It follows that for some process $q$, $k < q$.

Alternatively, suppose that $k > m$. We will show that the preconditions of Property 14 hold, and thus for some process $q$, $k < q$. Process $k$ does not enter the CS, as that would contradict $m$’s maximality. Therefore, process $k$ invokes $Abort(k)$ and, in turn, $Remove(k)$. Consider the process $p$ that wrote $go[k] = true$. $p$ executes $FindNext(v)_p$ and receives $k$. From the definition of $m, v \leq m$. If $v < m < k$, then Property 9 implies that $m$ invoked $Abort$, which is a contradiction. Therefore, $v = m$ and $p$ executes $FindNext(m)_p$, receiving $k$ in response. From Property 7, $FindNext(m)_p \rightarrow Remove(k)$. $Process k$ executes $Remove(k)$ and then $FindNext(m)_k$, receiving $b$. Therefore $FindNext(m)_k \rightarrow FindNext(m)_k$. If $b \notin \{\top, \perp\}$, then from Property 11, $k \leq b$. From Corollary 8, $b \neq k$. Thus, $b > k$ and writes $go[b] = true$, contradicting $k$’s maximality. If $b = \perp$, Property 10 implies that $i$ invokes $Remove(i)$, which is a contradiction. Therefore $b = \top$. Consider any $d$ such that $m < d < k$. By Property 9, $Remove(d) \rightarrow FindNext(m)_p$. We also have $FindNext(m)_p \rightarrow Remove(k)$. Therefore, $Remove(d) \rightarrow Remove(k)$. The preconditions of Property 14 thus hold.
Claim 19 shows that \( k <_{m} r \) for some \( q \). By Corollary 13, there exists a maximal \( r \) such that \( k <_{m} r \). The definition of \( <_{m} r \) implies that \( \text{Remove}(r) \) is invoked in \( E \) and that \( \text{FindNext}(m)_{k} \vdash \text{Remove}(r) \). Therefore, at time to when \( m \) sets \( \text{LastExited} = m \), \( \text{Remove}(r) \) has not yet completed. Once \( \text{Remove}(r) \) returns, \( r \) thus reads \( \text{Head} = \text{LastExited} = m \) at line 13 of Algorithm 3.3 and invokes \( \text{FindNext}(m)_{r} \), receiving some value \( c \).

If \( c = \bot \), this again implies \( i \) invokes \( \text{Remove}(r) \), which is a contradiction. If \( c = T \), let \( d \) such that \( m < d < r \). We have \( u < d < \max(k,r) \) and \( k <_{m} r \), and so by the definition of \( <_{m} r \), \( \text{Remove}(d) \vdash \text{Remove}(r) \). The preconditions for Property 14 thus hold for \( m \) and \( r > m \). This implies that for some process \( r \neq r \), \( r <_{m} r \), contradicting \( r \)'s maximality.

Finally, suppose that \( c \notin \{T,\bot\} \). If \( c > k \) then process \( r \) writes \( go[c] = \text{true} \), contradicting \( k \)'s maximality. Therefore \( c \leq k \). From condition 3 in the definition of \( <_{m} r \), we have \( \text{FindNext}(m)_{k} \vdash \text{Remove}(r) \). So \( \text{Remove}(k) \vdash \text{FindNext}(m)_{k} \vdash \text{Remove}(r) \vdash \text{FindNext}(m)_{r} \), and therefore \( \text{Remove}(k) \vdash \text{FindNext}(m)_{r} \). From Corollary 8, we have that \( \text{FindNext}(m)_{r} \) does not return \( k \), and therefore \( c < k \). From Property 6, we have \( c > m \). So \( m < c < \max(k,r) \). Condition 4 in the definition of \( <_{m} r \) implies that \( \text{Remove}(c) \vdash \text{Remove}(r) \). Therefore, \( \text{Remove}(c) \vdash \text{FindNext}(m)_{r} \). From Corollary 8, \( \text{FindNext}(m)_{r} \) does not return \( c \), which is a contradiction. \( \square \)

### 5.4 Complexity Analysis

The one-shot algorithm performs \( O(1) \) RMRs in addition to the RMRs performed by Tree operations. Each Tree operation is wait-free and takes \( O(\log_{W} N) \) steps, and thus at most \( O(\log_{W} N) \) RMRs. Given an execution \( E \), let \( R \) denote the number of processes that invoke \( \text{Remove}(r) \) in \( E \). In the following, some proofs are relegated to the full version [3] due to space limitations.

**Claim 20.** The RMR cost of \( \text{Remove}(r) \) is \( O(\log_{W} R) \).

Denote by \( R_{p}(t) \) the number of processes \( r \geq p \) that invoke \( \text{Remove}(r) \) in \( E_{t} \).

**Claim 21.** The RMR cost of \( \text{AdaptiveFindNext}(p)_{q} \) is \( O(\log_{W} R_{p}(t)) \), where \( t \) is the time in which \( \text{AdaptiveFindNext}(p)_{q} \) completes.

Proof. Consider an execution of \( \text{AdaptiveFindNext}(p) \) by process \( q \) that performs \( i > 2 \) iterations of the loop at line 44 of Algorithm 4.3. Consider the execution of \( \text{HasZeroToTheRight} \) at line 49 in iteration \( i-1 \), and let node be the node \( q \) accesses at this iteration. The check at line 45 guarantees that the value of the \( o \)ffset argument is less than \( W-1 \), and that \( p \notin \text{Child}(\text{node},\ W-1) \) (i.e., \( p \) is not the rightmost child). Since process \( q \) does not break at this iteration, we have that it receives \( \text{false} \) from this hasZeroToTheRight. This implies that the least significant (rightmost) bit of node.value is 1. It follows from the remove invariant that for all \( r \in \text{Child}(\text{node},\ W-1) \), process \( r \) invokes \( \text{Remove}(r) \) before \( \text{AdaptiveFindNext}(p)_{q} \) completes at time \( t \), i.e., in \( E_{t} \). Because \( p \notin \text{Child}(\text{node},\ W-1) \), for every \( r \in \text{Child}(\text{node},\ W-1) \), \( r > p \). There are \( W^{1-2} \) leaves in the subtree rooted at \( \text{Child}(\text{node},\ W-1) \). The number of processes \( r \geq p \) that invoke \( \text{Remove}(r) \) in \( E_{t} \) is \( R_{p}(t) \). Therefore, \( W^{1-2} < R_{p}(t) \), and so \( l = 2 + \log_{W} R_{p}(t) \). Each iteration of the loop performs a constant number of RMRs, and the claim follows. \( \square \)

**Corollary 22.** Each successful passage of the one-shot algorithm incurs \( O(\log_{W} A_{1}) \) RMRs, where \( A_{1} \) is the number of processes that abort during the passage. An aborted attempt incurs \( O(\log_{W} A_{1}) \) RMRs, where \( A_{1} \) is the number of processes that abort during the entire execution.

### 6 FROM ONE-SHOT TO LONG-LIVED LOCK

We present a generic transformation that converts a one-shot abortable lock algorithm \( L \), with space complexity \( s(N) \) (where \( s(N)2^{W} = O(1) \)) into a long-lived algorithm with the same asymptotic RMR cost as \( L \), and space complexity \( O(N \cdot s(N) + N^{2}) \). Our transformation preserves starvation-freedom, but not FCFS.

Figure 5 presents the pseudo-code of the transformation. For simplicity, we assume a system with unbounded word and memory size, in which allocating a new (and initialized) instance of the one-shot lock \( L \) is free of charge. Section 6.2 removes these assumptions.

The long-lived lock uses an instance of \( L \) to solve mutual exclusion, and dynamically switches to new instances to keep processes from accessing the same one-shot lock instance twice. We represent the long-lived lock as a tuple \( (\text{Lock}, \text{Spn}, \text{RefCnt}) \), where \( \text{Lock} \) is a pointer to the instance of \( L \), \( \text{Spn} \) is a pointer to a spin node associated with this instance, which contains a single boolean field \( go \), and \( \text{RefCnt} \) is a \( \lfloor \log N \rfloor \)-bit reference count, indicating the number of processes currently accessing the one-shot lock instance. The tuple is stored in a single memory word \( \text{LockDsc} \), which enables its fields to be manipulated atomically, as follows.

We store \( \text{RefCnt} \) in the lower bits of \( \text{LockDsc} \), allowing processes to use F&A to increment/decrement it while simultaneously obtaining a snapshot of the tuple. To acquire the lock, a process \( p \) uses F&A on \( \text{LockDsc} \) to increment \( \text{RefCnt} \), obtaining the instance \( l \) pointed to by \( \text{Lock} \) in response (line 62), which \( p \) then attempts to acquire. Once \( p \) finishes its attempt (either due to aborting or after releasing \( l \)), it uses a F&A on \( \text{LockDsc} \) to decrement \( \text{RefCnt} \) (line 70). If \( p \) decrements \( \text{RefCnt} \) to 0, it uses a CAS to atomically switch \( \text{Lock} \) and \( \text{Spn} \) to point to new one-shot lock and spin node instances, conditioned on \( \text{RefCnt} = 0 \) (line 76).

However, \( p \) will fail to switch the lock from \( l \) if another process increments \( \text{RefCnt} \) between lines 70 and 76. This scenario can lead to \( p \)'s next acquisition attempt occurring while \( \text{Lock} \) still points to \( l \), but \( p \) cannot access the one-shot lock \( l \) again. We use the spin node \( \text{spn} \) associated with \( l \) to efficiently prevent processes from accessing \( l \) again: \( p \) saves \( \text{spn} \) in a local variable when its attempt to acquire \( l \) finishes (line 70), and in the next acquisition attempt, \( p \) busy waits on \( \text{spn.go} \) if \( \text{LockDsc.Spns} = \text{spn} \) (lines 57–59). Once the lock is switched from \( l \), \( \text{spn} \) to new instances, \( \text{spn.go} \) is set (line 77), signalling the processes busy waiting on \( \text{spn.go} \) that \( \text{LockDsc.Lock} \neq l \), and so they may attempt to acquire the one-shot lock. Using the spin nodes thus establishes that \( \text{LockDsc.Lock} \) changed in \( O(1) \) RMRs. Without them, it would require accessing \( \text{LockDsc} \), which could incur \( N-1 \) RMRs, as \( \text{RefCnt} \) can change \( N \) times before \( \text{LockDsc.Lock} \) changes.

### 6.1 Correctness of the Transformation

Here, we prove:

**Theorem 23.** Let \( L \) be a one-shot starvation-free abortable lock. Our transformation yields a long-lived starvation-free abortable lock.
The long-lived algorithm satisfies starvation freedom.

**Lemma 27.** The long-lived algorithm satisfies mutual exclusion.

**Proof.** Assume towards a contradiction that processes \( p, q \) are both in the CS at time \( t \). Prior to entering the CS of the one-shot lock, both processes execute the F&A at line 62 and obtain LockDesc values \((l_p, s_p, ...)\) and \((l_q, s_q, ...)\). WLOG, assume that \( p \) performs its F&A first, at time \( t_0 \). Since \( p \) is in the CS at time \( t \), Claim 24 implies that LockDesc.Lock = \( l_p \) at time \( t \). Therefore, \( l_q = l_p \), as \( q \) performs its F&A at time \( t_1 \), \( t_0 < t_1 < t \). Both processes are thus in the same CS at time \( t \), contradicting mutual exclusion of the one-shot lock. \( \Box \)

**Lemma 27.** The long-lived algorithm satisfies starvation freedom.

**Proof.** Let \( E \) be an execution in which no process crashes outside the remainder section and every process that enters the CS eventually leaves it. Suppose that some process \( p \) is spinning at line 59 at time \( t \). Let \( l \) be the one-shot lock instance pointed to by \( p \)'s oldSpn variable at time \( t \). Then at some time \( t' < t \), \( p \) executed a F&A at line 70 that returned \((l, \_, \_1)\). Claim 24 implies that if \( p \) executes a F&A at line 70 that returns \((l, \_, \_1)\), then the previous execution of line 63 by the process invoked \( l.\text{Enter}() \). Therefore, Claim 25 implies that each process executes a F&A at line 70 that returns \((l, \_, \_1)\) at most once. Let \( q \) be the last process to execute such a F&A. Since \( q \) is last, this F&A returns \((l, \_, 1)\), and so eventually \( q \) executes the CAS at line 76. This CAS cannot fail because some process increments LockDesc.Refcnt (while LockDesc.L = \( l \)) as such a process will later call Cleanup() and execute line 70, in contradiction. Therefore, either \( q \)'s CAS succeeds or it fails because another process executes a CAS at line 76 that succeeds. In either case, eventually some process sets oldSpn.go to true, and so eventually \( p \) breaks out of its spin loop and invokes Enter() on some instance \( l' \) of the one-shot lock. Starvation-freedom of \( l' \) implies that if \( p \) does not abort, it will enter the CS. \( \Box \)

**Algorithm 6.2 Exit()**

1. \((l, s, r) \rightarrow \text{LockDesc}\)
2. \(l.\text{Exit}()\)
3. \(\text{Cleanup()}\)

**Algorithm 6.3 Cleanup()**

1. \((\text{oldLock}, \text{oldSpn}, \text{refcnt}) \rightarrow \text{F&A(\text{LockDesc}, 1)}\)
2. \(\text{if refcnt} = 1 \text{ then}\)
3. \(\text{newLock} \leftarrow \text{AllocateOneShotLockInstance()}\)
4. \(\text{newSpn} \leftarrow \text{AllocateSpinNode()}\)
5. \(\text{old} \leftarrow (\text{newLock}, \text{newSpn}, 0)\)
6. \(\text{if CAS(\text{LockDesc}, old, new)} \text{ then}\)
7. \(\text{oldSpn.go} \leftarrow \text{true}\)

**Figure 5:** Transformation of a one-shot abortable lock algorithm \( l \) into a long-lived abortable lock

**Claim 24.** Suppose process \( p \) invokes \( l.\text{Enter}() \) (line 63). Let \((l, s, r)\) be the LockDesc value \( p \) obtains at line 62 prior to this \( l.\text{Enter}() \) call. Then LockDesc remains \((l, s, r)\) until \( p \) executes the F&A at line 70.

**Proof.** Every process increments and decrements LockDesc.Refcnt once per acquisition attempt (lines 62 and 70, respectively). Therefore, LockDesc.Refcnt \( \geq 1 \) from \( p \)'s F&A execution at line 62 prior to invoking \( l.\text{Enter}() \) and until it executes the F&A at line 70 afterwards (after either aborting or releasing \( l \)). Now, LockDesc.Spun and LockDesc.Lock change only if a CAS at line 76 succeeds. However, a CAS at line 76 can succeed only if LockDesc.Refcnt = 0. The claim follows. \( \Box \)

**Claim 25.** Let \( E \) be an execution of the long-lived algorithm. Then no process executes \( l.\text{Enter}() \) twice on the same one-shot lock object.

**Proof.** Assume towards a contradiction that processes \( p, q \) are both in the CS at time \( t \). Prior to entering the CS of the one-shot lock, both processes execute the F&A at line 62 and obtain LockDesc values \((l_p, s_p, ...)\) and \((l_q, s_q, ...)\). WLOG, assume that \( p \) performs its F&A first, at time \( t_0 \). Since \( p \) is in the CS at time \( t \), Claim 24 implies that LockDesc.Lock = \( l_p \) at time \( t \). Therefore, \( l_q = l_p \), as \( q \) performs its F&A at time \( t_1 \), \( t_0 < t_1 < t \). Both processes are thus in the same CS at time \( t \), contradicting mutual exclusion of the one-shot lock. \( \Box \)

**6.2 Bounding Space and RMR Complexity**

We augment the long-lived lock with memory management schemes that safely recycle one-shot lock and spin node instances, so that the overall number of objects used by the algorithm is \( O(N) \) one-shot locks and \( O(N^2) \) spin nodes. These bounds enable maintaining LockDesc in a \( W \)-bit memory word, assuming that \( W = O(\log N) \). We only provide an overview of the schemes, due to space constraints. Details appear in the full version [3].

**Recycling one-shot locks.** A process that successfully installs a new one-shot lock instance holds on to the instance \( l \) that it replaced, and uses \( l \) to satisfy its next allocation. (This is safe because other processes attempt to acquire \( l \) only if LockDesc.Lock points to it, which is not the case after switching to a new one-shot lock instance.) The main problem is how to reset the variables of \( l \) so that when \( l \) gets reused they contain their initial values, but without having a single reset operation that incurs \( s(N) \) RMRs. To this end, we use a lazy reset scheme, in which the processes reset the variables of \( l \) as they are accessed inside the one-shot algorithm.

Our scheme borrows ideas from the scheme of Aghazadeh et al. [1, § 4], but does not “steal” bits from the words being reset. The idea of their scheme is to add a version number for each word \( w \) of \( l \), which is incremented each time \( l \) is acquired, and to encode a version number in each of \( l \)'s memory words, enabling processes to detect “stale” values. A process reading value \( x \) from a word uses \( x \) only if the version encoded in it equals \( l \)'s current version; otherwise, the process proceeds as if it read the word’s initial value. A process that writes to a word augments the write with \( l \)'s current version. In our scheme, for each word \( w \) of \( l \) we maintain a word \( V_w \), which contains a pair \((v_w, b_w)\), where \( v_w \) is \( w \)'s version and \( b_w \) is an incarnation bit. We also maintain two words, \( w_0 \) and \( w_1 \), and guarantee that \( w \)'s next incarnation, \( w_{1-b_w} \), always contains \( w \)'s initial value. When a process \( p \) first needs to access \( w \), it reads \((v_w, b_w)\) from \( V_w \). If \( v_w = v \), \( v_0 \) from \( V_w \) to \((v, 1-b) \) and resets \( w_0 \) to \( w \)'s initial value. It will subsequently access \( w_{1-b} \) whenever it needs to access \( w \).
To prevent wraparound of \( l \)'s version from making a stale word appear valid, we (like Aghazadeh et al. [1, §4]) additionally reset \( s(N)/2^W \) of \( l \)'s words each time \( l \) gets reused, which guarantees that after \( l \)'s version wraps around, each word has been reset. In summary, our scheme (1) increases the one-shot lock’s space complexity from \( s(N) \) to \( O(s(N)) \); (2) adds \( O(1) \) RMRs to the first access a process makes in the one-shot algorithm; and (3) adds \( O(s(N)/2^W) \) RMRs to the cost of allocating a one-shot lock instance.

Recycling spin nodes. Since a spin node might be accessed by a busy waiting process even after LockDesc no longer points to it, recycling spin nodes requires a safe memory reclamation scheme that recycles a spin node only if no process busy waits on it. We use the memory reclamation scheme of Aghazadeh et al. [2], which is similar to hazard pointers [22] but has constant worst-case RMR cost. Each process allocates spin nodes from a local pool. The reclamation scheme replenishes the pool as follows: A process \( p \) that switches LockDesc.Spn to point from spin node spin to another node restores the node spin. Once the scheme ascertains that no process is busy waiting on spin, it places spin into \( p \)'s pool. The reclamation scheme guarantees that a process can have at most \( N \) spin nodes that it retired but have not yet been placed into its pool. We therefore guarantee that pools never become empty by using pools of size \( N + 1 \). Overall, we manage \( O(N^2) \) spin nodes, and the reclamation scheme requires \( O(N) \) words for its shared variables.

The following result follows from the above discussion.

**Claim 28.** If \( l \) has RMR cost \( t(A_1, A_2) \) and space complexity \( s(N) \), and if \( s(N)/2^W = O(1) \), then the long-lived lock has RMR cost \( O(t(A_1, A_2)) \) and space complexity \( O(N \cdot s(N) + N^2) \).

### 7 RELATED WORK

Randomized abortable locks can obtain \( O(\log N) \log \log N \) expected RMR cost [23] or \( O(1) \) expected amortized RMR cost [12]. Both algorithms are for the CC model; use reads, writes, and CAS; and work against an adversary that is slightly weaker than the strong adaptive adversary, which can make scheduling decisions based on all past events, including the latest coin-flips.

Lee [20] obtains an \( O(\log N) \)-RMR (non-adaptive) abortable lock by modifying Yang and Anderson’s lock [26], making each two-process lock in their binary tree abortable. It is thus not clear that his construction can be generalized to make the tree \( W \)-ary. Jayanti’s adaptive \( O(\log N) \)-RMR abortable lock [17] hinges on a binary tree-based \( f \)-array [16]. While the \( f \)-array can be made \( W \)-ary, the RMR complexity of such a \( W \)-ary tree of height \( H = \log_W N \) is \( O(W \cdot H) \), which is not sublogarithmic. In contrast, with F&A we can aggregate certain information about a node’s children with one RMR, whereas the LL/SC-based \( f \)-array requires \( O(*) \) children RMRs. Further, unlike prior work, we simplify the problem to a one-shot variant, which we then (generically) transform to a long-lived lock.

### 8 CONCLUSION

Applying the transformation of Section 6 to the one-shot abortable lock of Section 3 yields a starvation-free (but not FCFS) abortable lock with space complexity \( O(N^2) \), in which the RMR cost of a successful passage is \( O(\log W A_1) \) and the RMR cost of an aborted attempt is \( O(\log_W A_2) \). Assuming \( W = \Theta(\log N) \), the worst-case RMR cost of the composed lock is \( O(\frac{N \log N}{\log \log N}) \). This paper thus shows that, with mutual exclusion, abortable locks can leverage additional primitives (beyond read, write and CAS) to obtain sublogarithmic worst-case RMR cost.

Several interesting questions remain. Our algorithm is for the CC model; the problem in the DSM model remains open. Our algorithm achieves \( O(1) \) RMR cost only if \( W = \Theta(\log N) \), whereas lock algorithms such as the MCS lock [21] obtain \( O(1) \) RMR cost with \( W = \Theta(\log N) \). Is this an inherent difference between abortable locks and regular locks? Finally, Jayanti’s \( O(\log N) \)-RMR abortable lock [17] satisfies FCFS and is adaptive to point contention; our algorithm does not have these properties. Can this gap be bridged?

### ACKNOWLEDGMENTS

This research was funded in part by the Israel Science Foundation (grant 2005/17). Adam Morrison is supported by Len Blavatnik and the Blavatnik Family Foundation.

### REFERENCES


