On the Complexity of Reader-Writer Locks*

[Extended Abstract]

Danny Hendler
Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel
hendlerd@cs.bgu.ac.il

ABSTRACT

A reader-writer lock [7] is a widely-used variant of the mutual exclusion lock abstraction [10]. It is shared by $n$ readers and $m$ writers, whose accesses of the Critical Section (CS) must satisfy the following requirement: reader processes are allowed to be in the CS simultaneously but each writer process requires exclusive access. We study the (worst-case) remote memory reference (RMR) complexity of reader-writer locks in the cache-coherent (CC) read/write model [2].

The tight logarithmic RMRs lower bound on mutual exclusion locks [6, 11] implies an $\Omega(\log m)$ lower bound on the RMR complexity of writers. But how does the RMR complexity of reader-writer locks depend on $n$, the number of readers? This is the question that we address in this work.

We establish an $\Omega(\log n)$ RMR complexity lower bound that holds even for single-writer reader-writer locks. It is derived from the following complexity tradeoff that we prove: if the number of RMRs incurred by the entry section of the writer is $O(f(n))$, then the RMR complexity of the reader’s exit section is $\Omega(\log \frac{n}{f(n)})$. The tradeoff holds even if processes may use the compare-and-swap (CAS) operation in addition to reads and writes. We present a family of reader-writer lock algorithms that establishes that the tradeoff is asymptotically tight for any function $f(n)$.

Keywords

Reader-writer lock; Remote memory references; Compare and swap

1. INTRODUCTION

Over the last 20 years, shared-memory mutual exclusion research has investigated the remote memory references (RMR) complexity of local-spin mutual exclusion algorithms [2]; much of this work has focused on read/write mutual exclusion [10] (see [1, 3, 4, 8, 16, 17, 19, 21] for some examples).

A reader-writer lock [7] is a widely-used variant of the mutual exclusion lock abstraction. It is shared by $n$ reader processes and $m$ writer processes. While readers are allowed to be in the Critical Section (CS) simultaneously, no process is allowed in the CS concurrently with a writer. Reader-writer locks must also satisfy the Concurrent Entering requirement [13, 18]: if all writers are in the remainder section, a reader must be able to enter the CS in a bounded number of its own steps.

There exist read/write mutual exclusion algorithms that incur only a logarithmic number of RMRs per passage (entry and corresponding exit of the critical section) [21] and a matching lower bound established that this is optimal [6, 11]. Less is known about the RMR complexity of reader-writer lock algorithms. The logarithmic RMRs lower bound on mutual exclusion locks [6, 11] implies an $\Omega(\log m)$ lower bound on the RMR complexity of writers. But how does the RMR complexity of reader-writer locks depend on $n$, the number of readers? This is the question that we address in this work.

We prove the following RMR complexity tradeoff for reader-writer locks: If the number of RMRs incurred by the entry section of the writer is $O(f(n))$, then the RMR complexity of the reader’s exit section is $\Omega(\log \frac{n}{f(n)})$. The tradeoff holds even if processes may use the compare-and-swap (CAS) operation in addition to reads and writes and even for single-writer reader-writer locks. It implies that at least one of these complexities must be asymptotically logarithmic in $n$. In turn, this implies that the RMR complexity of reader-writer locks is $\Omega(\max(\log n, \log m))$.

We also present a family $A_f$ of reader-writer lock algorithms, using read, write and CAS operations, that is parameterized on $f$ — the number of RMRs performed by writers. For all $f(n)$, the RMR complexities of writers and readers executing a passage of $A_f$ are $\Theta(f(n))$ and $\Theta(\log(n/f(n)))$, respectively. This establishes that the tradeoff is tight for all $f(n)$.

The rest of this paper is organized as follows. In Section 2, we describe the model and define the reader-writer lock problem. We present our complexity tradeoff in Section 3. This is followed by Section 4, where the algorithms matching

---

*Supported in part by the Israel Science Foundation (grant 1749/14) and by the Lynne and William Frankel Center for Computer Science.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PODC'16, July 25-28, 2016, Chicago, IL, USA
© 2016 ACM. ISBN 978-1-4503-3964-3/16/07...$15.00
DOI: http://dx.doi.org/10.1145/2933057.2933099

1The lower bound of [6] applies also to mutual exclusion algorithms that may use comparison primitives, such as compare-and-swap, in addition to reads and writes.
the tradeoff are described. We prove the correctness of the algorithms in Section 5. We conclude, in Section 6, with a short discussion of related work and directions for future research.

2. MODEL

We assume a standard model of an asynchronous shared memory system, where a set \( P = \{R_1, \ldots, R_n, W_1, \ldots, W_m\} \) of \( n \) reader processes and \( m \) writer processes communicate via a set \( V \) of shared variables. In each step \( s \), a process applies a read, write, or compare-and-swap (CAS) operation to a shared-memory variable \( v \), and returns some response \( res \). \( \text{CAS}(v, \text{expected}, \text{new}) \) changes the value of variable \( v \) to \text{new} only if its value just before it is applied is \text{expected}; it returns the value of \( v \) prior to its application.

We say that step \( s \) accesses \( v \). If \( s \) applies a read or CAS operation to \( v \), we say that \( s \) is a reading step. Thus, a CAS step is both a reading and a writing step. We say that a step is trivial, if it does not change the value of the variable which it accesses. If a step is not trivial, we say it is non-trivial.

An execution fragment \( E \) is a sequence of steps taken by processes as they perform their algorithms starting from some configuration \( C \). We say that \( E \) is permissible from \( C \) and denote its execution from \( C \) by \( C \rightarrow E \). We let \( CE \) denote the configuration reached after executing fragment \( E \) starting from \( C \). An execution \( E \) is an execution fragment that starts from the initial configuration, denoted by \( C_{\text{init}} \). We let \( \bot \) denote the empty execution fragment.

We consider the cache-coherent (CC) shared-memory model. In the CC model, each processor maintains copies of shared variables inside its private cache, whose consistency is ensured by a coherence protocol. Our results apply to both the write-through and write-back CC coherence protocols [20]. Quoting from [12]: "In a write-through protocol, to read a variable \( v \) a process \( p \) must have a (valid) cached copy of \( v \). If it does, \( p \) reads that copy without causing an RMR; otherwise, \( p \) causes an RMR that creates a cached copy of \( v \). To write \( v \), \( p \) causes an RMR that invalidates (i.e., effectively deletes) all other cached copies of \( v \) and writes \( v \) to main memory. In a write-back protocol, each cached copy is held in either shared or exclusive mode. To read a variable \( v \), a process \( p \) must hold a cached copy of \( v \) in either mode. If it does, \( p \) reads that copy without causing an RMR. Otherwise, \( p \) causes an RMR that: (a) eliminates any copy of \( v \) held in exclusive mode, typically by downgrading the status to shared and, if the exclusive copy was modified, writing \( v \) back to memory; and (b) creates a cached copy of \( v \) held in shared mode. To write \( v \), \( p \) must have a cached copy of \( v \) held in exclusive mode. If it does, \( p \) writes that copy without causing RMRs. Otherwise, \( p \) causes an RMR that: (a) invalidates all other cached copies of \( v \) and writes any modified copy held in exclusive mode back to memory; and (b) creates a cached copy of \( v \) held in exclusive mode."

2.1 Reader-Writer Locks

A reader-writer lock allows concurrent access of the critical section for read-only operations, while write operations require exclusive access. It is a relaxation of a mutual exclusion lock, which only allows a single process in the critical section at any given time. Each process sharing the read-write lock is labeled a reader or a writer. A reader-writer lock algorithm consists of an entry section and an exit section that surround a critical section (CS). A single execution of the entry, critical and exit section is called a passage. When a process is not in the midst of performing a passage, we say that it is in the remainder section. In \( C_{\text{init}} \), the initial configuration, all processes are in their remainder section and all shared variables store their initial values. We say that a configuration \( C \) is quiescent if all processes are in the remainder section in \( C \).

We say that a process \( p \) fail-stops in an infinite execution \( E \), if \( p \) enters the CS and does not subsequently exit from it or if \( p \) has an incomplete passage in \( E \) and executes only a finite number of steps in \( E \). We make the standard assumption that processes do not fail-stop outside the remainder section.

We now state the properties required from a reader-writer lock.

- **Mutual Exclusion**: If a writer is in the CS at any given time, then no other process is in the CS at that time.

- **Bounded Exit**: A process completes the exit section in a bounded number of its own steps.

- **Deadlock Freedom**: Infinitely many passages complete in any execution in which processes perform infinitely many steps.

- **Concurrent Entering**: Let \( C \) be a configuration in which all writers are in the remainder section and let \( C \rightarrow E \) be an execution fragment in which no writers take steps. Then there is a constant \( b \) such that any reader that executes at most \( b \) steps in \( C \rightarrow E \) in the course of its entry section enters the CS.

3. A COMPLEXITY TRADEOFF FOR READER-WRITER LOCKS

In executions in which only writer processes participate, a read-write lock algorithm functions as a mutual-exclusion lock. Consequently, read-write locks from read, write and CAS operations are subject to the logarithmic RMR lower bound of Attiya et al. [6]. This implies a lower bound on RMR complexity of writer processes that is logarithmic in \( m \) — the number of writers.

In this section, we prove an \( \Omega(\log n) \) RMR complexity lower bound that holds even for single-writer read-write locks. In fact, we prove a more general complexity tradeoff between the number of RMRs incurred by the exit section of a reader and the number of RMRs incurred by the entry section of a writer. This tradeoff implies that at least one of these complexities must be asymptotically logarithmic in \( n \), the number of readers.

3.1 High-Level Proof Overview

We now provide a high-level overview of our proofs. We construct an execution whose structure is depicted by Figure 1. It is comprised of 3 execution fragments. In fragment \( E_1 \), all readers execute their entry section and enter the CS. In fragment \( E_2 \), all readers return to the remainder section. Finally, in fragment \( E_3 \), a single writer, say \( W_1 \), enters the CS.

The key intuition behind our complexity tradeoff is the following. Before entering the CS, the writer must verify that
the CS is empty. In order to do so, it must “become aware” of all the readers that participated in $E_2$. If the writer accesses at most $f(n)$ shared variables in its entry section, then the value of each such variable must, on average, “inform” the writer about at least $n/f(n)$ readers that completed their exit section. Say the writer reads value $v$, then it follows that $v$ must have been written by a reader that (on average) “became aware” of $\Theta(n/f(n))$ other readers that exited the CS. To obtain our tradeoff, we schedule reader steps in $E_2$ so that they collect this “awareness” relatively slowly.

Our proofs require a formalization of the notion of knowledge. Attiya and Hendler defined the concepts of process awareness sets and variable familiarity sets [5]. The awareness set of process $p$ quantifies the number of processes that $p$ may be aware of in an execution. The familiarity set of variable $v$ quantifies the number of processes a process may become aware of by reading $v$.

We extend this formalism in two ways. First, we define the new notion of an expanding step — a step that expands the awareness set of the executing process; we prove that each such step incurs an RMR. Second, we generalize these notions so that we can argue about the number of readers that a reader becomes aware of during its exit section only, that is, after it “forgets”, in a sense, the information it collected during its entry section. Technically this is done by defining the awareness/familiarity sets of execution fragments rather than of executions only.

Execution $E_2 = \sigma_0 \sigma_1 \ldots \sigma_r$, for some $r \geq 0$, is constructed in $r + 1$ iterations. Each iteration $i \in \{1, \ldots, r\}$ begins with expanding steps executed by all the readers not yet in the remainder section (see Figure 1). To establish the tradeoff, we prove that $r \in \Theta(\log (n/f(n)))$ must hold.

### 3.2 Detailed Proofs

We start by formalizing a notion of knowledge. Let $C \hookrightarrow E$ be an execution fragment. We quantify the knowledge that processes and variables hold regarding which processes take steps in $E$. The awareness set of process $p$ after $C \hookrightarrow E$, denoted $AW(p, C \hookrightarrow E)$, is the set of processes of whose participation in $C \hookrightarrow E$ may become aware via its reading steps in $E$. Similarly, the familiarity set of variable $v$ after $C \hookrightarrow E$, denoted $F(v, C \hookrightarrow E)$, is the set of processes whose participation in $C \hookrightarrow E$ may be inferred by reading $v$. The following two (mutually recursive) definitions formalize these notions.

**Definition 1** Let $C \hookrightarrow E$ be an execution fragment and $v$ be a variable. If no non-trivial steps were applied to $v$ in $C \hookrightarrow E$, then we let $F(v, C \hookrightarrow E) = \emptyset$. Otherwise let $s$ be the last such step, executed by process $p$, and let $E = E_1sE_2$. The familiarity set of $v$ after $C \hookrightarrow E$, denoted $F(v, C \hookrightarrow E)$, is defined as follows.

1. If $s$ is a write step, we let $F(v, C \hookrightarrow E) = AW(p, C \hookrightarrow E_1)$.
2. If $s$ is a CAS step, we let $F(v, C \hookrightarrow E) = AW(p, C \hookrightarrow E_1) \cup F(v, C \hookrightarrow E_1)$.

Definition 1 implies that the familiarity set of a variable $v$ depends on the last non-trivial step $s$ that accesses $v$. If $s$ is a write step, executed by some process $p$, then $v$’s familiarity set becomes the awareness set of $p$ (when it applies it), since $s$ overwrites $v$. If $s$ is a CAS step, then $v$’s familiarity set is extended by $p$’s awareness set. This is because a successful CAS step applied to a variable increases, in general, the knowledge that may be inferred from reading that variable. The familiarity set of a variable to which no non-trivial steps were applied is empty.

**Definition 2** Let $C \hookrightarrow E$ be an execution fragment and $p$ be a process. The awareness set of $p$ after $C \hookrightarrow E$, denoted $AW(p, C \hookrightarrow E)$, is defined as follows.

1. If $E$ contains no reading steps by $p$, we let $AW(p, C \hookrightarrow E) = \{p\}$.
2. Otherwise, let $s$ be the last reading step by $p, v$ be the variable accessed by $s$ and $E = E_1sE_2$. We let $AW(p, C \hookrightarrow E) = AW(p, C \hookrightarrow E_1) \cup F(v, C \hookrightarrow E_1)$.

We say that process $p$ is $C \hookrightarrow E$-aware of process $q$ if $q \in AW(p, C \hookrightarrow E)$ holds.

Intuitively, a process $p$ is aware of the participation of another process $q$ in an execution fragment if there is (either direct or indirect) information flow from $q$ to $p$ in that fragment via shared memory. Thus, whenever $p$ executes a reading step (a read or a CAS) to some variable $v$, the processes in $f$’s familiarity set (if any) are added to $p$’s awareness set. Notice that the awareness-set of a process can only grow as an execution fragment unfolds.

The following observations are immediate from Definition 1 and 2.

**Observation 1** Let $C \hookrightarrow E$ and $C \hookrightarrow F$ be two execution fragments. If $E$ is a prefix of $F$, then $AW(p, C \hookrightarrow E) \subseteq AW(p, C \hookrightarrow F)$ for all $p \in P$.

**Observation 2** Let $C \hookrightarrow E$ be an execution fragment and $v$ be a variable. Assume there were non-trivial steps applied to $v$ in $C \hookrightarrow E$ and let $s$ be the last such step, applied by some process $p$. Also let $E = E_1sE_2$. Then $F(v, C \hookrightarrow E_1) = AW(p, C \hookrightarrow E_1)$.

**Proof.** If $s$ is a write step, this follows from the first condition of Definition 1. If $s$ is a CAS step, this follows from the second condition in Definitions 1 and 2 and the fact that a CAS step is both a reading and a writing step.

Our proofs use the notion of expanding steps, which are steps that expand the awareness sets of processes. This is formalized by the following definition.

**Definition 3** Let $C \hookrightarrow E$ be an execution fragment, where $E = E_1sE_2$. We say that $s$ is a $C \hookrightarrow E$ expanding step, if there exists a process $p$ such that $|AW(p, C \hookrightarrow E_1)| < |AW(p, C \hookrightarrow E_2)|$.

**Fact 1** Let $C \hookrightarrow E$ be an execution fragment, where $E = E_1sE_2$, and assume that $s$ is a $C \hookrightarrow E$ expanding step, applied by some process $p$, that accesses some variable $v$, then: $|AW(q, C \hookrightarrow E_1)| < |AW(q, C \hookrightarrow E_2)| \Rightarrow s$ is a reading step and $q = p$.

We now show that every expanding step is an RMR.

**Lemma 1** Let $s$ be a $C \hookrightarrow E$ expanding step by some process $p$, then $p$ incurs an RMR when it executes $s$. 
Proof. Let $E = E_1 s E_2$. Since $s$ is a $C \rightarrow E$ expanding step, there is a process $q$ such that $|AW(q, C \rightarrow E_1)| < |AW(q, C \rightarrow E_1 s)|$. From Fact 1, $s$ is a reading step and $q$ is not. If $p$ does not have a valid copy of $v$ in its cache immediately after $C \rightarrow E_1$, then $p$ incurs an RMR when it executes $s$. Assume otherwise towards a contradiction. Let $t'$ be the last step previously executed by $p$ (either in $E$ or in the execution leading to $C$), before which there was no valid copy of $v$ in $p$'s cache. Let $t'$ and $t$ respectively denote the times when $s'$ and $s$ are executed. Thus, $p$'s cache has a valid copy of $v$ in all its cache. Let $t$ be an expanding step, from Definitions 1-3, there must be a step in the execution leading to $s$ that access a specific variable $v$, if there are any. Let $s_j$ be the first of these steps and let $s_f$ be a process that executes $s_j$. There are two cases to consider.

1. If $v$ was written to in $s_2$, then its value after $E_1 s_2$ differs from the expected parameter of $s_j$ and of all the other CAS steps of $s_3$. Thus, all these steps are trivial, hence:

$$F(v, C \rightarrow E_1 s_2) = F(v, C \rightarrow E_1) + F(v, C \rightarrow E_2) \leq 2 \cdot M(C \rightarrow E_1),$$

where the last inequality follows from Inequality 3.

Moreover, for every $p \in P$ that takes a step in $s_3$:

$$AW(p, C \rightarrow E_1 s_2) \leq AW(p, C \rightarrow E_1) + F(v, C \rightarrow E_2) \leq 2 \cdot M(C \rightarrow E_1),$$

where the last inequality follows again from Inequality 3.

2. If $v$ was not written to in $s_2$, then $s_j$ is non-trivial and does change the value of $v$, making all subsequent CAS steps non-trivial. Hence:

$$F(v, C \rightarrow E_1 s_2) = F(v, C \rightarrow E_1) + F(v, C \rightarrow E_2) \leq 2 \cdot M(C \rightarrow E_1).$$

Moreover, for every $p \in P$ that takes a step in $s_3$:

$$AW(p, C \rightarrow E_1 s_2) \leq AW(p, C \rightarrow E_1) + F(v, C \rightarrow E_2) \leq 3 \cdot M(C \rightarrow E_1).$$

The lemma now follows from Inequalities 2, 4, 6 and 8.

Lemma 3 Let $C \rightarrow E$ be an execution fragment and let $p \in P$ be a process that executes steps in an arbitrary order. Let $s_1$ denote the resulting sequence of steps. No step in $CE \rightarrow s_1$ changes the value of the variable to which it is applied, hence:

$$\forall v \in V : F(v, C \rightarrow E_1) = F(v, C \rightarrow E) \leq M(C \rightarrow E),$$

where the last inequality follows from Definition 1 and Observations 1 and 2.

Moreover, for every $p \in P$ that has a step in $s_1$ accessing variable $v_p$:

$$AW(p, C \rightarrow E_1) \leq AW(p, C \rightarrow E) + F(v, C \rightarrow E) \leq 2 \cdot M(C \rightarrow E).$$

We next order all remaining write steps (in an arbitrary order) and let $s_2$ denote the resulting sequence of steps. Consider all the steps of $s_2$ that access a specific variable $v$ and let the last of them be denote by $l_v$. Let $p_i$ be the process that executes $l_v$. From Definition 1:

$$F(v, C \rightarrow E_1 s_2) = AW(p_i, C \rightarrow E) \leq M(C \rightarrow E).$$

Moreover, for every $p \in P$ that takes a step in $s_2$:

$$AW(p, C \rightarrow E_1 s_2) = AW(p, C \rightarrow E) \leq M(C \rightarrow E).$$

Finally, we schedule all remaining CAS steps (in an arbitrary order) and denote the resulting sequence by $s_3$. We also let $s = s_1 s_2 s_3$. Consider all steps of $s_3$ that access a specific variable $v$, if there are any. Let $s_j$ be the first of these steps and let $p_j$ be a process that executes $s_j$. There are two cases to consider.

Proof. Let $s_h \in E'$ be a step by process $q$ such that $E' = E_1 s_h E_2$. Since $s_h \in E$ we can write $E = E_1 s_h E_2$. If $|E_1| = k$ we denote $E_1$ by $s_k$ and $E_2$ by $s_k'$. By induction on $k = 1, \cdots |E'|$, we prove that $C \rightarrow s_k'$ is an execution fragment. For the base case, $s_0 = \bot$ and $C \rightarrow \bot$ is an execution fragment.
Assume now that \( C \mapsto \sigma_k \) is an execution fragment and let \( \sigma_{k+1} = \sigma_k s_q \), we prove that \( C \mapsto \sigma_{k+1} \) is an execution fragment. Assume otherwise towards a contradiction. From Observation 1, \( E' \) is obtained from \( E \) by removing the suffixes of some processes’ steps in \( E \). If \( \sigma_k \) is an execution but \( \sigma_{k+1} \) is not, it must be that \( s_q \) is a reading event from some variable \( v \) that returns some response \( res \) (the value it returns in \( C \mapsto E \)) but \( v \)’s value after \( C \mapsto E \sigma_k \) is some \( res' \neq res \). This implies in turn that there is a writing event \( w \) that writes \( res \) to \( v \) in \( C \mapsto E \) before \( v \) is read by \( s_q \), and \( w \) is removed from \( E \). It follows that \( w \) was executed by a process that was aware of \( p \) when it executed \( w \). From Definitions 1 and 2, this implies that \( s_q \) should have been removed from \( E \) as well. This is a contradiction. \( \square \)

Let \( A \) be a reader-writer lock algorithm. Our proofs construct an execution \( E = E_1 E_2 E_3 \) of \( A \) defined as follows. In \( E_1 \), all reader processes, \( R_1, \ldots, R_n \), leave the remainder section, execute the entry section and enter the critical section. Let \( C_1 \) be the resulting configuration. In execution fragment \( C_1 \mapsto E_2 \), all the readers execute the exit section and return to the remainder section. Let \( C_2 \) denote the resulting configuration. Finally, in execution fragment \( C_2 \mapsto E_3 \), writer process \( W_1 \) performs its entry section and enters the critical section.

**Observation 3** Execution \( E \) is a finite, well-defined execution and \( W_1 \) is in the critical section after \( E \).

**Proof.** Since all writers are in the remainder section in \( C_{init} \), Concurrent Entering ensures that \( R_1, \ldots, R_n \) complete their exit section in a bounded number of steps, so \( E_1 \) is finite. Bounded Exit ensures that \( R_1, \ldots, R_n \) each complete its exit section in a bounded number of steps, so \( C_{init} E_1 \mapsto E_2 \) is finite. From Construction 2, \( C_2 \) is quiescent, so Deadlock Freedom ensures that \( C_{init} E_1 E_2 \mapsto E_3 \) is finite and \( W_1 \) is in the CS after \( E \). \( \square \)

We now show that while \( W_1 \) performs its entry section, it must become aware of steps executed by all the readers as they performed their exit section.

**Lemma 4** \( \{R_1, \ldots, R_n\} \subseteq AW(W_1, C_1 \mapsto E_2 E_3) \).

**Proof.** From Observation 3, \( E \) is well-defined and finite. Assume towards a contradiction that there is a process \( R_o \notin AW(W_1, C_1 \mapsto E_2 E_3) \). We construct from \( E_2 E_3 \) a sequence of events \( E' \) as follows: We remove from \( E_2 E_3 \) all the steps executed by \( R_o \) as well as all the steps executed by other processes when they are aware of \( R_o \). From Lemma 3, \( C_1 \mapsto E' \) is an execution.

Since \( R_o \) is in the critical section in \( C_1 \) and since it takes no steps in \( E \mapsto E' \), it is in the critical section also after \( C_1 \mapsto E' \). \( W_1 \) is in the critical section after \( E \). Since \( W_1 \) is not aware of \( R_o \) at the end of \( E \), none of its steps is removed from \( E_1 E_2 \). Hence it is inside the critical also at the end of \( C_1 \mapsto E', \) together with \( R_o \). This violates Mutual Exclusion. \( \square \)

**Theorem 5** Let \( A \) be a reader-writer lock algorithm for \( n \) readers and a single writer, using read, write and CAS operations. If the RMR complexity of the writer’s exit code is \( O(f(n)) \), then the RMR complexity of the reader’s exit code is \( \Omega\left(\frac{\log(n)}{f(n)}\right)\).

**Proof.** Let \( E_1 \) be an execution in which all reader processes, \( R_1, \ldots, R_n \), leave the remainder section, execute the entry section and enter the critical section. Let \( C_1 \) be the resulting configuration. We iteratively construct execution fragment \( C_1 \mapsto E_2, E_2 = \sigma_0 \sigma_1, \ldots, \sigma_r \), in which all readers execute their exit code and return to the remainder section. For \( j \in \{0, \ldots, r\} \), we let \( E'_j = \sigma_0, \sigma_1, \ldots, \sigma_j \), so \( E_2 = E'_r \). Our construction maintains the following two invariants for all \( j \in \{1, \ldots, r\} \):

1. \( \forall v \in V : |F(v, C_1 \mapsto E'_j)| = 3^j \forall p \in P : |AW(p, C_1 \mapsto E'_j)| = 3^j \).
2. All readers not in their remainder section after \( C_1 \mapsto E'_j \) are about to execute an expanding step.

For the base case, from \( C_1 \mapsto \ firing, \) we let all readers execute their steps until they are either about to perform an expanding step or they return to the remainder section. We let \( \sigma_0 \) denote the resulting execution fragment and \( E'_{0} = \sigma_0 \). From Bounded Exit, \( \sigma_0 \) is finite. The second invariant follows immediately from construction. As for the first invariant, we have \( \forall v \in V : |F(v, C_1 \mapsto \emptyset)| = 0 \) and \( \forall p \in \{R_1, \ldots, R_n\} : |AW(p, C_1 \mapsto \emptyset)| = 1 \), so \( \max_{p \in P} |AW(p, C_1 \mapsto \emptyset)| = 1 \). Since none of the steps in \( \sigma_0 \) is expanding, \( \forall p \in \{R_1, \ldots, R_n\} : |AW(p, C_1 \mapsto \emptyset)| = 1 \) hence, from Observation 2, \( \forall v \in V : |F(v, C_1 \mapsto \emptyset)| \leq 1 \) and so the invariant holds.

Assume we have constructed \( E'_j \). Let \( R_j \) denote the set of readers that did not complete their exit section during \( C_1 \mapsto E'_j \). If \( R_j = \emptyset \), the construction of \( E_2 \) terminated and so \( r = j \). Otherwise, we construct \( E'_{j+1} \) as follows. From the second invariant applied to \( E'_j \), each of the readers of \( R_j \) is about to perform an expanding step. From Lemma 2, there is an expanding \( \sigma'_{j+1} \) of these steps, such that:

\[
\max_{p \in P} |AW(p, C_1 \mapsto E'_j \sigma'_{j+1})| \leq 3^{max_{p \in P} |AW(p, C_1 \mapsto E'_j)|} \leq 3^{j+1},
\]

where the last inequality follows from the first invariant applied to \( E'_j \). Next, we let all the readers of \( R_j \) take steps until they either return to the remainder section or are about to perform another expanding step. We denote the resulting execution fragment by \( \sigma_{j+1} \). From Bounded Exit, \( \sigma_{j+1} \) is finite. We let \( \sigma_{j+1} = \sigma_{j+1} \sigma'_{j+1} \) and \( E'_{j+1} = E'_j \sigma_{j+1} \). The second invariant follows immediately from the construction of \( \sigma_{j+1} \). Since none of the steps of \( \sigma'_{j+1} \) is expanding, none of the awareness sets grows during \( \sigma_{j+1} \) so the first invariant also holds.

We proceed with the construction of \( E_2 \) until either all readers return to the remainder section or until we complete \( \log_3(n/f(n)) \) iterations (that is, until we construct \( E'_n \)), whatever occurs first. We let \( C_2 \) denote the configuration immediately after \( C_1 \mapsto E_2 \).

Next, we show that \( r = \Omega(\log_3(n/f(n))) \). Assume otherwise towards a contradiction, then \( r \in o(\log_3(n/f(n))) \) holds. From construction and the second invariant applied to \( E'_r \):

\[
\forall v \in V : |F(v, C_1 \mapsto E_2)| = 3^r = o(n/f(n)). \tag{9}
\]

Let \( C_2 \mapsto E_3 \) be an execution fragment in which writer \( W_1 \) enters its critical section as it runs solo starting from \( C_2 \). \( E_3 \) is finite, since all the readers are in the remainder section in \( C_2 \) and from Deadlock Freedom. From assumption, the number of RMRs incurred by \( W_1 \) in the course of its entry
section is $O(f(n))$. From Lemma 1, this implies that $W_1$ executes $O(f(n))$ expanding steps. From Equation 9, this implies in turn that $\vert AW(W_1, C_1 \leftrightarrow E_2 E_3) \vert < n$. This contradicts Lemma 4.

We have shown that $r = \Omega(\log_2(n/f(n)))$. Let $R_t$ be a reader that did not complete its exit section in the course of $E_{t+1}$. There must be such a reader, as otherwise the construction would have stopped before iteration $r$. From construction, $R_t$ executes exactly $r$ expanding events during $C_1 \leftrightarrow E_2$ as it exits the critical section. The theorem now follows from Lemma 1.

Corollary 6 Let $A$ be a reader-writer lock algorithm for $n$ readers and a single writer, using read, write and CAS operations. Then either the RMR complexity of the writer’s entry code is $\Omega(\log(n))$ or the RMR complexity of the reader’s exit code is $\Omega(\log(n))$.

Corollary 7 Let $A$ be a reader-writer lock algorithm for $n$ readers and $m$ writers, using read, write and CAS operations. Then its RMR complexity is $\Omega(\max(\log m, \log n))$.

4. A FAMILY OF ALGORITHMS MATCHING THE TRADEOFF

In this section, we show that the complexity tradeoff we presented in Section 3 is tight. We do so by presenting a family $A_f$ of reader-writer lock algorithms, using read, write and CAS operations, that is parameterized on $f$ — the number of RMRs performed by writers. For all $f(n)$, the RMR complexities of writers and readers executing a passage of $A_f$ are $\Theta(f(n))$ and $\Theta(\log(n/f(n)))$, respectively. This establishes that the tradeoff is tight for all (non-superlinear) $f(n)$.

The pseudo-code is given in Algorithm 1. We start by describing the shared variables used by it. We (statically) partition the readers to groups of size $K = \lfloor n/f(n) \rfloor$ each, according to their ID. The processes in each of these $f(n)$ groups consolidate information using $K$-process counter objects (see line 1). A counter supports atomic `add` and `read` operations. Jayanti [15] presented an $f$-array based counter implementation from read, write and load link/store conditional (LL/SC) operations, where add and read operations perform logarithmic and constant numbers of steps, respectively. Jayanti’s construction is easily modified to use CAS instead of LL/SC [14]. Thus, in our algorithm a reader performs an add operation in $O(\log K)$ steps and a writer may read all group counters in $O(f(n))$ steps.

The $i$'th readers group (for $i \in \{0, \ldots, f(n) - 1\}$) uses counters $C[i]$ and $W[i]$. $C[i]$ counts the number of readers that are currently in a passage; it is incremented in line 31 when a reader starts a passage and decremented in line 39 in the exit section. $W[i]$ counts the number of readers that observed a writer and are waiting to be signalled by it. A group-$i$ reader increments $W[i]$ in line 34 after it observes the writer (in line 32) and decrements $W[i]$ in line 37 after it is signalled by the writer and stops waiting.

Writers ensure mutual exclusion among themselves by using $WL$ (see line 2) — an $m$-process starvation-free read/write mutual exclusion lock algorithm satisfying Bounded Exit. There are such algorithms with logarithmic per-passage RMR complexity (e.g. [21]). Each passage by a writer process is associated with a unique sequence number. This sequence

```
<table>
<thead>
<tr>
<th>Pseudo-code for reader-writer algorithm $A_f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 const $K = \lceil n/f(n) \rceil$; shared $C[f(n)]$, $W[f(n)]$:</td>
</tr>
<tr>
<td>$K$-process Counter init $[0, \ldots, 0]$;</td>
</tr>
<tr>
<td>2 shared $WL$: $m$-process mutex lock;</td>
</tr>
<tr>
<td>3 shared $WSEQ$: int init 0;</td>
</tr>
<tr>
<td>4 shared $WSIG[f(n)]:$ int init $&lt; 0, 0, \perp &gt;$, $RSIG$: int</td>
</tr>
<tr>
<td>init $&lt; 0, NOP &gt;$;</td>
</tr>
<tr>
<td>5 Procedure ReaderPassage($i$);</td>
</tr>
<tr>
<td>6 WL.Enter();</td>
</tr>
<tr>
<td>7 for $i \in {0, \ldots, f(n) - 1}$ do</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9 end</td>
</tr>
<tr>
<td>10;</td>
</tr>
<tr>
<td>11 $RSIG \leftarrow WSEQ, PREENTRY &gt;$;</td>
</tr>
<tr>
<td>12 for $i \in {0, \ldots, f(n) - 1}$ do</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17 end</td>
</tr>
<tr>
<td>18 $RSIG \leftarrow WSEQ, WAIT &gt;$;</td>
</tr>
<tr>
<td>19 for $i \in {0, \ldots, f(n) - 1}$ do</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>23 end</td>
</tr>
<tr>
<td>24 Critical Section;</td>
</tr>
<tr>
<td>25 $WSEQ \leftarrow WSEQ + 1$;</td>
</tr>
<tr>
<td>26 $RSIG \leftarrow WSEQ, NOP &gt;$;</td>
</tr>
<tr>
<td>27 WL.Exit();</td>
</tr>
<tr>
<td>28;</td>
</tr>
<tr>
<td>29 Procedure ReaderPassage($i$);</td>
</tr>
<tr>
<td>30 local $i$: int init $[p/f(n)]$;</td>
</tr>
<tr>
<td>31 $C[i].add(1)$;</td>
</tr>
<tr>
<td>32 $&lt; seq, op &gt; \leftarrow RSIG$;</td>
</tr>
<tr>
<td>33 if $op = \perp$ then</td>
</tr>
<tr>
<td>34</td>
</tr>
<tr>
<td>35</td>
</tr>
<tr>
<td>36</td>
</tr>
<tr>
<td>37</td>
</tr>
<tr>
<td>38 end</td>
</tr>
<tr>
<td>39 Critical Section;</td>
</tr>
<tr>
<td>40 $C[i].add(-1)$;</td>
</tr>
<tr>
<td>41 $&lt; seq', op' &gt; \leftarrow RSIG$;</td>
</tr>
<tr>
<td>42 if $op' = \perp$ then</td>
</tr>
<tr>
<td>43</td>
</tr>
<tr>
<td>44</td>
</tr>
<tr>
<td>45</td>
</tr>
<tr>
<td>46 end</td>
</tr>
<tr>
<td>47 else if $op = \perp$ then</td>
</tr>
<tr>
<td>48</td>
</tr>
<tr>
<td>49</td>
</tr>
<tr>
<td>50 Procedure HelpWCS(int seq);</td>
</tr>
<tr>
<td>51 if $C[i].read() = W[i].read()$ then</td>
</tr>
<tr>
<td>52</td>
</tr>
<tr>
<td>53 end</td>
</tr>
<tr>
<td>54 }</td>
</tr>
</tbody>
</table>
```
number is stored in variable \( WSEQ \) (line 3), which is incremented by a writer before it returns to the remainder section. Readers and writers synchronize by signalling each other. A writer uses variable \( RSIG \) (line 4) to signal readers and group-\( i \) readers use \( WSIG[i] \) (line 4) to signal the writer. \( RSIG \) holds a pair of values. The first is a sequence number identifying a writer process passage. The second is either \( NOP \) — indicating that no writer currently holds \( WL \), or an opcode signalling the readers that they may need to perform an operation on behalf of the writer that holds \( WL \). We now provide a detailed description of the algorithm, in which the usage of these variables is also explained.

**Writers pseudo-code**

Writers perform procedure \( WriterPassage \) (lines 5–28) to execute a passage. After capturing \( WL \) in line 6, process \( q \) initializes the variables it uses for synchronizing with concurrent readers. First, it sets variable \( WSIG[i] \), for \( i \in \{0, \ldots, f(n) – 1\} \), used by by group-\( i \) readers to signal the writer, to values-pair \( < WSEQ, \bot > \) (lines 7–9). This is the initial state of \( WSIG[i] \) for signalling the writer performing the passage identified by sequence number \( WSEQ \). Then, it sets \( RSIG \) to a pair of values \( < WSEQ, PREENTRY > \) (line 11). The second value — \( PREENTRY \), is an opcode instructing a reader-\( i \) process that reads \( 0 \) from \( C[i] \) to notify \( q \) (via \( WSIG[i] \)). The purpose of the \( PREENTRY \) command, as well as that of the loop of lines 12–17, is to verify that no readers are already waiting (for previous writer passages), before \( w \) instructs concurrent readers to wait for its current passage. This is done as follows. For each readers-group \( i \in \{0, \ldots, f(n) – 1\} \), \( q \) reads \( C[i] \) (line 13). If it is positive, \( q \) busy-waits to be signalled on \( WSIG[i] \) (line 14). As we prove, in this case, one of the readers that read \( 0 \) from \( W[i] \) is guaranteed to successfully CAS \( WSIG[i] \) to values-pair \( < wseq, PROCEED > \) (in line 45), where \( wseq \) is the current value of \( WSEQ \). At the end of the \( i \)’th loop iteration, \( w \) ensures that \( WSIG[i] \) has value \( < WSEQ, PROCEED > \), in preparation for issuing a new command.

Having verified that no readers are waiting, \( q \) must ensure that readers in their passage either exit or will observe it and will wait until it completes its current CS entry. It does by setting \( RSIG = < WSEQ, WAIT > \) in line 18. Now, any reader that reads \( RSIG \) in line 32 must busy-wait on it (in line 36) until its value is changed by \( q \). Next, \( q \) ensures that all the readers that did not observe it in their entry section clear the CS (lines 19–23). For each readers-group \( i \), it reads \( C[i] \) (line 20) and, if non-zero, busy-waits (line 21) until some group-\( i \) reader signals it (in line 52). This completes the entry section.

In its exit section, writer \( q \) increments \( WSEQ \) (in line 25) and initializes \( RSIG \) for the next writer CS entry (in line 26). The latter operation signals readers that already started waiting that they may proceed to the CS. Finally, \( w \) releases the writers lock (line 27).

**Readers pseudo-code**

Readers perform procedure \( ReaderPassage \) (lines 29–49) to execute a passage. A group-\( i \) reader \( p \) first indicates it starts a passage by incrementing \( C[i] \) (line 31). It then reads \( RSIG \) - the variable using which the writer that holds \( WL \) signals the readers, returning a pair of values to local variables \( seq \) and \( op \) (line 32). To ensure that readers wait for it, a writer writes the \( WAIT \) command (along with the sequence number associated with its passage) in line 18. If \( op \neq WAIT \), \( p \) may safely enter the CS. Otherwise, let \( q \) be the writer holding lock \( WL \), then \( p \) must wait for \( q \), so it increments the number of group-\( i \) waiting processes (line 34).

It then calls the \( HelpWCS \) procedure (line 35). \( HelpWCS \) (lines 50–54) reads counters \( C[i] \) and \( W[i] \). If both reads return the same value then, as we prove, it is safe for \( q \) to enter the CS (if it is not there already), so \( p \) attempts to signal \( q \) that it may do so (line 52) by CAS'ing \( WSIG[i] \).

It may be the case that multiple readers attempt to signal \( q \). As we prove, the semantics of CAS and the fact that value written by it must match the sequence number \( seq \) associated with \( w \)’s passage ensure that exactly one reader succeeds in signalling \( q \). After \( HelpWCS \) returns, \( p \) waits until it is signalled on variable \( RSIG \) that \( q \) completed this passage (line 36). As we prove, it is now safe for \( p \) to enter the CS. Before it does, \( p \) decrements \( W[i] \) in line 37.

In its exit section, reader \( p \) first decrements \( C[i] \) in line 40. Then, it reads \( RSIG \) once again in line 41. If the \( op \) field equals \( PREENTRY \), then a writer \( w \) whose passage sequence number is \( seq \) previously requested (in line 11) to be signalled when counter \( C[i] \) becomes \( 0 \). In this case, \( p \) checks if \( C[i] \) equals \( 0 \) (in line 43) and, if so, attempts to signal \( w \) (in line 45) that it may proceed in its entry section. Otherwise, if \( op \) equals \( WAIT \), then \( p \) calls the \( HelpWCS \) procedure (described previously) in line 48, in case \( w \) should be signalled to enter the CS (in its passage associated with the sequence number \( seq \)).

---

5. **ALGORITHM CORRECTNESS PROOFS**

In this section, we show that algorithms \( A_t \) are correct and provide the following properties: Mutual Exclusion, Bounded Exit, Deadlock Freedom and Concurrent Entering. We also prove that readers do not starve. We start by introducing some notation. For process \( p \) and integer \( N \), we write \( pc_p = N \) if the next line that \( p \) is about to execute is line number \( N \). We write \( pc_p \in [N_1, N_2] \), if the next line that \( p \) will execute is some line \( N, N_1 \leq N \leq N_2 \). If \( pc_p \in \{35, 48\} \), then \( p \) is about to invoke procedure \( HelpWCS \). Thus, if \( pc_p \in [N_1, N_2] \) and \( 35 \in [N_1, N_2] \) or \( 48 \in [N_1, N_2] \), then \( pc_p \in [51 – 54] \cup [N_1, N_2] \) is implicitly assumed (that is, the body of procedure \( HWCS \) is part of the lines-range). For local variable \( v \) and process \( p \), we let \( val_p(\tau) \) denote the value of the \( p \)’s local instance of \( v \). For shared variable \( V \), we let \( val(\upsilon) \) denote \( V \)’s value.

Definition 4 We say that reader \( r \) is waiting, if \( pc(r) \in [34, 36] \). We say it is waiting for writer passage seq, if it is waiting after reading values-par < seq, X > from RSIG in line 32.

Definition 5 We say that writer \( w \) is waiting, if \( pc(w) = 14 \) or \( pc(w) = 21 \).

Observation 4 Mutual exclusion between writer processes is maintained.

Proof. Immediate from the fact that writer processes start their entry section by capturing an \( m \)-process mutual exclusion lock (in line 6) and end it by releasing it (in line 27).
Lemma 8 If a writer is in the CS, then no reader is in the CS.

Proof. Assume towards a contradiction that some writer \( w \) is in the CS together with a group-\( i \) reader \( r \). Assume \( w \) is executing writer passage \( u \). Consider the \( i \)th iteration of the loop of lines 19–23 performed by \( w \) before it entered the CS. Assume first that \( w \) reads 0 from \( C[i] \) in line 20, then \( r \) did not yet execute line 31 in its entry section and so it will read a WAIT opcode from WSIG (in line 32) and must busy-wait (in line 36) until \( w \) exits the CS.

Assume that \( w \) reads a positive value from \( C[i] \) in time \( t \) and thus has to busy-wait in line 21. Since \( w \) eventually manages to proceed to the CS, it must eventually read values-pair \( < \text{seq}, \text{CS} > \) from WSIG\([i]\), where seq is the sequence number of its passage. This values-pair can only be written in line 52 of the HelpWCS procedure. Thus, from line 52, there is a time \( t' < t \) in which all readers that incremented \( C[i] \) and did not yet decrement it (if any) are waiting, and therefore may not enter the CS before \( w \) exits the CS and executes line 26. Moreover, any group-\( i \) reader that increments \( C[i] \) after \( t' \) and before \( w \) exits the CS must read a WAIT opcode from WSIG (in line 32) and will also need to wait. This is a contradiction. □

Lemma 9 Algorithms \( A_f \) satisfy Mutual Exclusion.

Proof. Follows immediately from Observation 4 and Lemma 8. □

Lemma 10 Algorithms \( A_f \) satisfy Bounded Exit.

Proof. The exit code of readers (lines 40–54) contains no await instructions or other loops. The same is true for the exit code of writers (lines 28), so the only place where the Bounded Exit property may be violated is the body of WL.exit, but this is not the case as \( WL \) is a lock that provides Bounded Waiting. □

Observation 5 Let \( C \) be a configuration in which all writers are in the remainder section, then the opcode stored in variable RSIG is NOP.

Proof. The claim holds in the initial configuration since RSIG is initialized to \( < \text{nop}, \text{nop}, \text{nop} > \). Since all writers are in the remainder section, the last writer passage (if there were any) was completed and so RSIG was set to \( < \text{seq}, \text{NOP} > \) (for some sequence number seq) in line 26. □

Lemma 11 Let \( w \) be a writer in its entry section. If \( pc(w) = 18 \), then there are no waiting readers.

Proof. From the code, reader processes only wait for a writer after reading an opcode of WAIT from the RSIG variable in line 32. Clearly from the code and from Observation 5, when \( pc(w) \in \{ 6, 18 \} \) holds, the opcode stored by RSIG is not WAIT. Thus, no reader starts to wait when \( pc(w) \in \{ 6, 18 \} \). It therefore suffices to show that any reader that waits when \( w \) starts its entry section no longer waits when \( pc(w) = 18 \).

Assume towards a contradiction that this is not the case and let \( t_2 \) be such a reader. So, denoting by \( t_3 \) the time when \( w \) starts its entry section and by \( t_2 \) the time when it finishes the loop of lines 12–17, \( r \) is waiting all throughout \( [t_1, t_2] \). Let \( i \) be the readers group to which \( r \) belongs, it follows that \( w \) evaluates the condition of line 13 in the \( i \)th loop iteration as true and proceeds to busy-wait in line 14. Since we assume that \( w \) reaches line 18, it must be that some group-\( i \) reader \( r' \) wrote values-pair \( < \text{seq}, \text{PROCEED} > \) to WSIG\([i]\), where seq is the sequence number identifying \( w \)'s current passage. Moreover, since WSEQ is incremented only in the writers exit section, it must be that \( r' \) read RSIG after \( w \) wrote it in line 11 of its current passage. Thus, \( r' \) read 0 from \( C[i] \) at some time \( t \in (t_1, t_2] \), implying that \( r' \) was no longer waiting in time \( t \) hence it is not waiting when \( pc(w) = 18 \). This is a contradiction. □

Lemma 12 Algorithms \( A_f \) satisfy Concurrent Entering.

Proof. Let \( C \) be a configuration in which all writers are in the remainder section and let \( C \rightarrow E \) be an execution fragment in which only readers take steps. Let \( r \) be a process that executes steps in its entry section in the course of \( E \). If \( r \) executes line 32 in \( E \) then, from Observation 5, it reads a NOP opcode from RSIG in line 32 and so enters the CS in a constant number of its steps. This is true also if \( r \) read a non-WAIT opcode in line 32 before \( C \). So assume that \( r \) read a \( < \text{seq}, \text{WAIT} > \) values-pair from RSIG before \( C \) and that \( r \) is waiting in \( C \). From Lemma 11, seq is the sequence number of the last writer passage, as otherwise the last writer cannot arrive at line 18. It follows that RSIG \( = < \text{seq}, \text{NOP} > \) all throughout \( C \rightarrow E \) and so \( r \) executes line 36 at most once in \( E \) and enters the CS in a constant number of its steps. □

Observation 6 For all \( i \in \{ 0, \ldots, f(n) - 1 \} \), \( C[i] > 0 \) if and only if there is a group-\( i \) reader \( r \) such that \( pc_r \in \{ 32, 40 \} \).

Proof. Follows from lines 31, 40 and their order. □

Lemma 13 Let \( w \) be a writer process executing writer passage seq and let \( C \) be a configuration in which \( pc_w \in \{ 12, 17 \} \). Let \( C \rightarrow E \) be an execution fragment from \( C \) in which, for each \( i \in \{ 0, \ldots, f(n) - 1 \} \), \( C[i] \) equals 0 in at least one configuration reached in the course of \( C \rightarrow E \) (not necessarily all in the same configuration), leading to configuration \( C' \).

Then there is a (possibly empty) finite solo execution fragment of \( w \), \( C' \rightarrow E' \), during which \( WSIG[i] = < \text{seq}, \text{WAIT} > \) holds for all \( i \in \{ 0, \ldots, f(n) - 1 \} \).

Proof. Consider \( w \)'s execution of iteration \( i \) of lines 12–17. If \( C[i].read() \) returns 0 in line 13, then \( w \) proceeds to write \( < \text{seq}, \text{WAIT} > \) to WSIG\([i]\) in line 16 before \( C \) or during \( C \rightarrow EE' \).

Otherwise, \( C[i].read() \) returns a positive value so \( w \) becomes waiting. Assume this occurs in time \( t \) and let \( t_2 > t \) be the time when \( C \rightarrow E \) completes. From line 11, \( RSIG = < \text{WSEQ}.\text{PREENTRY} > \) holds in \( t \) and so RSIG continues to assume this value as long as \( w \) does not execute line 18. Also, from assumptions, there is a time \( t_1, t < t_1 < t_2 \), during the execution of \( C \rightarrow E \) in which \( C[i] \) becomes 0. Thus, there is at least one reader that performs a CAS in line 45 during \( C \rightarrow E \) at least one successful CAS is applied to WSIG\([i]\), setting it to values-pair \( < \text{seq}, \text{PROCEED} > \). Then \( w \) stops waiting and proceeds to write \( < \text{seq}, \text{WAIT} > \) to WSIG\([i]\) in line 16. □

Lemma 14 Let \( w \) be a writer process executing writer passage seq and let \( C \) be a configuration in which \( pc_w \in \{ 19, 23 \} \). Let \( C \rightarrow E \) be an infinite execution fragment from \( C \). Then there is a finite prefix \( E' \) of \( E \) so that \( w \) is in the CS after \( C \rightarrow E' \).
Proof. Let $t$ be the time when $w$ sets $RSIG \leftarrow <seq.WAIT>$ in line 18 and configuration $C$ is reached. From lines 12–17, $WSIG[i]=<seq.PROCEED>$ holds for all $i \in \{0, \ldots, f(n)-1\}$ in $t$. Let $r$ be a group-$i$ reader and consider the value of its program counter in $t$. There are two cases to consider.

1. If $pc_r > 32$ then $r$ will not wait for $w$; from Lemma 11, $r$ is not waiting for any other writer passage in $C$. Consequently, from Lemmas 10 and 12, $r$ eventually returns to the remainder section. Before it does, it decrements $C[i]$ in line 40 at which point it is not counted by either $C[i]$ or $W[i]$.

2. Otherwise $pc_r \leq 32$, so $r$ eventually reads values-pair $<seq.WAIT>$ from $RSIG$ in line 32. It then increments $W[i]$ in line 34 at which point it is counted by both $C[i]$ or $W[i]$ before it proceeds to busy-wait in line 36.

It follows from the above that before all group-$i$ readers return to the remainder section or start to busy-wait in line 36, $W[i] = C[i]$ holds. Also, in both of the above cases, $r$ calls the $HelpWCS$ procedure. Thus, there is at least one group-$i$ reader that performs a CAS in line 52 and at least one successful CAS is applied to $WSIG[i]$, setting it to values-pair $<seq.CS>$. It follows that eventually writer $w$ stops busy-waiting and may proceed to the CS.

\[\Box\]

**Lemma 15** Algorithms $A_f$ satisfy Deadlock Freedom.

**Proof.** Assume towards a contradiction that the algorithm has an infinite execution fragment $C \hookrightarrow E$ in which no process enters the CS. The following cases exist.

- $C \hookrightarrow E$ is a fragment in which all writers are in the remainder section. This contradicts Lemma 12.
- Otherwise, since the writers synchronize on the WL lock, in $C \hookrightarrow E$ there is a single writer process $w$ such that $pc_w \in [7 - 27]$. If $w$ evaluates the conditions of line 13 and line 20 as false for all $i$ then it enters the CS in a finite number of steps. This is a contradiction.

Assume, then, that $w$ is waiting forever in $C$, so $pc_w = 14$ or $pc_w = 21$. In the first case, from Lemma 13, there is a (possibly empty) prefix of $C \hookrightarrow E$ after which $WSIG[i] = <seq.WAIT>$ for all $i \in \{0, \ldots, f(n)-1\}$. As only $w$ itself may write this value to $WSIG[i]$, it must be that $w$ stops busy-waiting. Otherwise, $w$ is waiting forever in 21. This, however, is a contradiction to Lemma 14.

\[\Box\]

We next prove that readers do not starve.

**Lemma 16** There is no reader starvation in Algorithms $A_f$.

**Proof.** Assume towards a contradiction that there is an infinite execution $E$ in which some reader $r$ is forever in the entry section. It must be that $r$ busy-waits in line 36. Let $t$ be the time when $r$ executes line 32. Since $r$ waits, it reads in time $t$ values-pair $<seq.WAIT>$, where $seq$ is some sequence number associated with a passage of some writer $w$. This implies that $w$ already executed line 18 in the passage associated with $seq$, so $pc_w > 18$ in $t$. From Lemma 14, there is a finite prefix $E'$ of $E$ so that $w$ is in the CS after $E'$. From Lemma 10, $w$ returns to the remainder in a bounded number of steps after that. Thus, $w$ eventually increments $WSEQ$ in line 25. Since the value of $WSEQ$ is non-decreasing, eventually $r$ reads in line 36 a values-pair that is different from $<seq.WAIT>$ and so proceeds to the CS. This is a contradiction.

\[\Box\]

**Lemma 17** The RMR complexities incurred by writer and reader processes as they perform a passage in algorithm $A_f$ are $O(f(n))$ and $O((\log n / f(n)))$, respectively.

**Proof.** A writer process $q$ performs $f(n)$ iterations in the loops of lines 7–9, lines 12–17, and lines 19–23. In each iteration line, possibly except for the lines in which $q$ awaits, it incurs a constant number of RMRs per iteration. It therefore remains to show that $q$ incurs only a constant number of RMRs in each await line. We now consider these lines.

In line 14, $q$ busy-waits on $WSIG[i]$, until it reads the PROCEED code. Before waiting, it initializes $WSIG[i]$ to $<val(WSEQ),\perp>$ in line 8. From the code of the readers and the semantics of the CAS operation, the value of $WSIG[i]$ can now be changed only by line 45 to $<val(WSEQ),PROCEED>$. It follows that $q$ incurs at most a single RMR in line 14.

Process $q$ busy waits also in line 21. Before busy-waiting, $q$ sets $WSIG[i]$ to $<val(WSEQ),WAIT>$ in line 16. From the code of the readers and the semantics of the CAS operation, the value of $WSIG[i]$ can now be changed only by line 52 to $<val(WSEQ),CS>$. It follows that $q$ incurs at most a single RMR in line 21. This concludes the proof for the writers.

A reader process $r$ incurs $O((\log n / f(n)))$ RMRs whenever it applies an add operations on a counter object, in lines 31, 34, 37 or 40. In all non-await lines, it only incurs a constant number of RMRs. It remain to consider line 36, which is the single await line in the reader’s code.

In line 36, $r$ waits until it reads a value of $RSIG$ other than $<val,seq),WAIT>$ and, once it does, stops waiting. Since the value of $WSEQ$ is incremented whenever a writer passage ends, the same value is never written twice to $RSIG$, so $r$ incurs at most two RMRs in line 36.

\[\Box\]

**Theorem 18** Algorithms $A_f$ guarantee there is no reader starvation and satisfy the following properties: Mutual Exclusion, Bounded Exit, Deadlock Freedom and Concurrent Entering. The RMR complexity of a reader passage is $O(f(n))$ and the RMR complexity of a reader passage is $O((\log n / f(n)))$.

**Proof.** Follows from Lemmas 9, 10, 12, 15, 16 and 17.

\[\Box\]

### 6. DISCUSSION

Hendler and Khait [14] presented complexity tradeoffs between read and update operations for several concurrent objects and use a proof technique similar to ours. The key differences between our lower bound proof technique and theirs are the following. First, whereas their proofs consider obstruction-free implementations and use the step complexity metric, ours consider blocking ones and use RMR complexity. Additionally, in order to capture the knowledge accumulated by readers in their exit section only,
our notion of knowledge must be applied to execution fragments rather than to executions (starting from the initial configuration).

A lower bound of Danek and Hadzilacos [9] implies an \( \Omega(n) \) RMRs lower bound on Distributed Shared Memory (DSM) [2] reader-writer locks. This linear bound does not apply to the CC model, however.

Hadzilacos and Danek [9] presented a 2-session group mutual exclusion algorithm for the CC model that can be easily transformed to a reader-writer lock algorithm with \( \Omega(\max(\log n, \log m)) \) RMR complexity. The algorithm uses fetch-and-add and compare-and-swap operations, in addition to reads and writes. Our reader-writer algorithms attain the same complexity using read, write, and compare-and-swap operations only; they also allow choosing one of many optimal tradeoff points between the reader and writer complexities.

Bhatt and Jayanti presented a constant-RMRs reader-writer lock algorithm for the CC model, using fetch-and-add, read and write operations.

Our algorithms use compare-and-swap, in addition to read and write. An interesting open question is whether there is an algorithm with the same properties and RMR complexity using read and write operations only.

Our algorithms guarantee that readers do not starve. Writers, however, may starve if there are always readers performing passages. Finding a family of reader-writer algorithms (implemented from the same operations) that match our complexity tradeoff and provide better fairness is left for future work.

7. REFERENCES