# MULTIPLE-WRITER COMPOSITE REGISTERS*

James H. Anderson†

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-89-26 September 1989
July 1990 (Revision)

## ABSTRACT

A *composite register* is an array-like variable that is partitioned into a number of components. An operation of such a register either writes a value to one of the components, or reads the values of all of the components. A composite register reduces to an ordinary atomic register when there is only one component. In a related paper, we showed that atomic registers can be used to implement a composite register in which there is only one writer per component. In this paper, we show that a composite register with multiple writers per component can be implemented from a composite register with one writer per component. Together, these two constructions show that it is possible for a process of a concurrent program to take an atomic snapshot of an entire shared memory without using mutual exclusion.

**Keywords:** Atomicity, atomic register, composite register, concurrency, interleaving semantics, linearizability, shared variable, snapshot.

**CR Categories:** D.4.1, D.4.2, F.3.1.

---

* A preliminary version of this paper will be presented at the *Ninth Annual ACM Symposium on Principles of Distributed Computing*.
† Work supported in part by Office of Naval Research Contract N00014-89-J-1913.

# 1 Introduction

The concept of an atomic register is of fundamental importance in the theory of concurrent programming; see, for example, [4, 5, 8, 9, 11, 12, 13, 14, 15]. An *atomic register* is a shared data object that can either be read or written (but not both) in one indivisible operation. Such a data object is characterized by the number of processes that can write it, the number of processes that can read it, and the number of bits that it stores. The simplest atomic register can be written by one process, read by one process, and store a one-bit value; the most complex can be written and read by several processes and store any number of bits. The previously cited papers show that the most complex atomic register can be implemented in terms of the simplest.

In [2], we defined a shared data object, called a *composite register*, that extends the notion of an atomic register. A composite register is an array-like variable that is partitioned into a number of components. An operation of such a register either writes a value to one of the components, or reads the values of all of the components. A composite register reduces to an ordinary atomic register when there is only one component.

We consider here the important question of whether atomic registers can be used to construct composite registers. Such a construction consists of a set of writer and reader programs that communicate via a set of "internal" atomic registers. A process writes a value to one of the components of the constructed composite register by invoking one of the writer programs for that component. A process reads the values of all of the components by invoking one of the reader programs. Different programs can be invoked by different processes concurrently; the net effect, however, is required to resemble that of a serial invocation. The programs are restricted to be wait-free, i.e., synchronization primitives and unbounded busy-waiting loops are not allowed. This restriction guarantees that a process reads or writes the constructed composite register in a finite amount of time, regardless of the activities of other processes. It also ensures that the read or write of a process is immune to the failure of other processes that also access the constructed composite register.

We use a two step approach to show that atomic registers can be used to construct composite registers. In [2], we showed that atomic registers can be used to construct a composite register in which there is only one writer per component. In this paper, we use the construction of [2] to implement a composite register in which several writers per component are allowed.

One of the surprising consequences of this result is that, using only atomic registers, a process of a concurrent program can take an atomic "snapshot" of an entire shared memory without using mutual exclusion. Such a shared memory can be implemented by a single composite register, with each shared variable corresponding to a component of the register. To write a given variable, a process writes the corresponding component of the composite register. To read some set of variables, a process reads the entire composite register, and then selects the values of the components corresponding to the set. A global snapshot

operation is performed by simply reading the set of all variables.

The problem of constructing a composite register from atomic registers has also been considered independently by Afek et al. [1, 10]. In particular, Afek et al. show that an "atomic snapshot" primitive can be constructed from multiple-writer atomic registers. It is interesting to note that the construction given in [2] uses only single-writer atomic registers. Thus, the construction given in this paper (which is based upon the one in [2]) gives us a means for implementing a composite register using only single-writer atomic registers. It follows, then, that our construction can be used to implement a multiple-writer atomic register (the case in which there is only one component) from single-writer atomic registers.

Composite registers are quite powerful and can be used to implement a number of interesting shared data objects. For example, as shown in [2], composite registers can be used to implement a shared variable that can be *either* read or incremented in one atomic step. This result is somewhat surprising because it has been shown both by Herlihy [6] and by Anderson and Gouda [3] that it is impossible to implement a shared variable that can be *both* read and incremented in one atomic step.

The rest of the paper is organized as follows. In Section 2, we formally define the problem of constructing a composite register from atomic registers. In Section 3, we present our construction along with its proof of correctness. The version of the construction that is considered in Section 3 has unbounded space complexity. We show in an appendix that this version can be transformed into one with bounded space complexity. Concluding remarks appear in Section 4.

# 2 Composite Register Construction

In this section, we give the conditions that a composite register construction must satisfy to be correct. For brevity, we will be rather informal about describing what we mean by a "construction." However, our treatment of the correctness condition will be formal.

**Terminology:** In order to avoid confusion, we henceforth capitalize terms such as "Read" and "Write" when they apply to the *constructed* composite register, and leave them uncapitalized when they apply to the internal variables of a construction. □

A construction consists of a set of Writer programs and a set of Reader programs that communicate via a set of "internal" variables. A Writer program is invoked in order to Write a value to a component of the constructed composite register. A Reader program is invoked in order to Read the values of all of the components of the constructed composite register. Each Writer program has one input parameter indicating the value to be Written; each Reader program has one output parameter for each component of the constructed register.

We designate a composite register construction by a 4-tuple $K/L/M/N$, where $K$ is the number of components, $L$ is the number of bits per component, $M$ is the number of

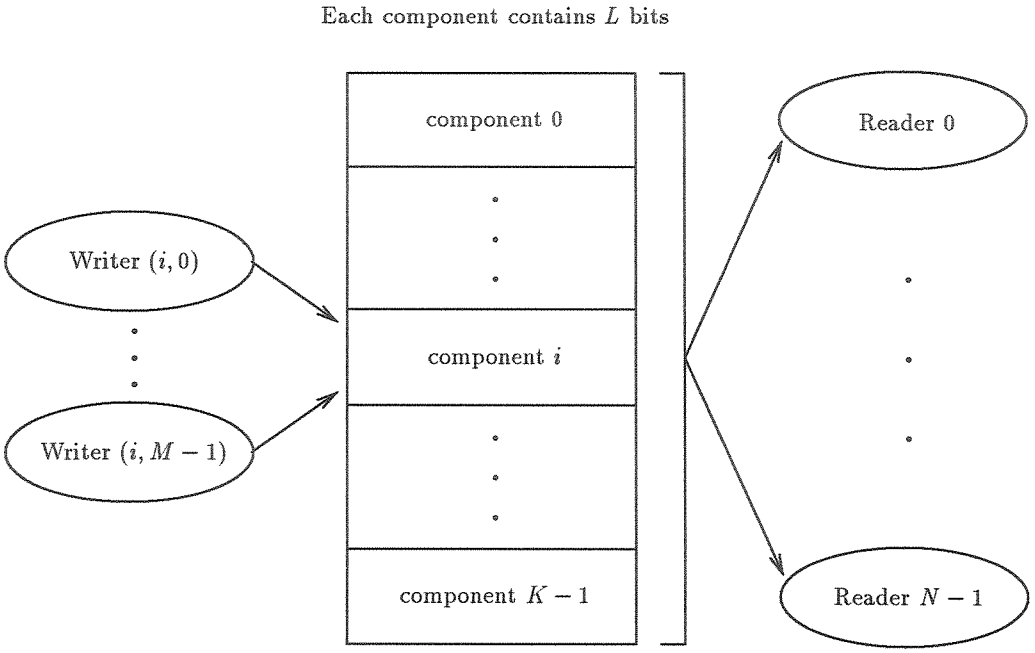


Figure 1: $K/L/M/N$ composite register structure.

Writers per component, and $N$ is the number of Readers. (Thus, a $1/L/M/N$ composite register is an ordinary atomic register.) The structure of a $K/L/M/N$ composite register construction is depicted in Figure 1. In this figure, ovals denote programs, boxes denote variables, and arrows denote direction of communication; an outgoing arrow from a program to a variable indicates that the program Writes the variable, while an arrow in the reverse direction indicates that the program Reads the variable. Note that this figure only depicts the Writer programs for component $i$. For an example of a Reader or Writer program, see the programs in Figures 4 and 5.

Each internal shared variable of a construction corresponds to an atomic register — thus, a statement of a program can either read a single shared variable, or write a single shared variable, but not both; i.e., in each statement, there is at most one occurrence of a shared variable. As mentioned in the introduction, each program of a construction is "wait-free," i.e., synchronization primitives and busy-waiting loops are not allowed. (A more formal definition of wait-freedom is given in [3].)

We now define several concepts that are needed to state the correctness condition for a construction. These definitions apply to a given construction.

**Definition:** A *state* is an assignment of values to the variables of the construction. One or more states are designated as *initial states*. □

**Definition:** An *event* is an execution of a statement of a program. □

**Definition:** Let $t$ and $u$ be any two states of a construction such that state $u$ is the result of executing some statement at state $t$. If $e$ is the event corresponding to this statement execution, then we write $t \xrightarrow{e} u$. A *history* of a construction is a sequence $t_0 \xrightarrow{e_0} t_1 \xrightarrow{e_1} \cdots$ where $t_0$ is an initial state. □

A Reader (or Writer) program can be repeatedly invoked to Read (or Write) the constructed composite register. Therefore, a given statement may be executed many times in a given history. Each such execution corresponds to a distinct event.

**Definition:** Event $e$ *precedes* another event $f$ in a history iff $e$ occurs before $f$ in the history. The set of events in a history corresponding to some program invocation is called an *operation*. An operation $p$ *precedes* another operation $q$ in a history iff each event of $p$ precedes all events of $q$. □

Observe that the precedes relation is an irreflexive total order on events and an irreflexive partial order on operations.

For the proof of correctness of a construction, it is sufficient to consider only histories that do not contain any incomplete program executions (i.e., incomplete operations). From now on, we deal only with such well-formed histories.

**Definition:** A Write operation of component $k$ of the constructed composite register, where $0 \leq k < K$, is called a *$k$-Write operation*. □

In order to avoid special cases when proving the correctness of a construction, we make the following assumption concerning the initial Write operations.

**Initial Writes:** For each $k$, where $0 \leq k < K$, there exists a $k$-Write operation that precedes each other $k$-Write operation and all Read operations. □

According to the following definition, if several operations are executed concurrently, then the net effect should be equivalent to some serial order. This definition is similar to the definition of linearizability as given in [7].

**Definition:** Let $h$ be a history of a construction. History $h$ is *atomic* iff the precedence relation on operations (which is a partial order) can be extended[1] to a total order $\sqsubset$ where for each Read operation $r$ in $h$ and each $k$ in the range $0 \leq k < K$, the value Read by $r$ for component $k$ is the same as the value Written by the $k$-Write operation $v$ that is defined as

[1] A relation $R$ over a set $S$ *extends* another relation $R'$ over $S$ iff for each $x$ and $y$ in $S$, $xR'y \Rightarrow xRy$.

follows: $v \sqsubset r \ \wedge \ \neg(\exists w : w \text{ is a } k\text{-Write} : v \sqsubset w \sqsubset r)$. □

Note that the Write operation $v$ in the definition above exists by our assumption concerning the initial Writes.

**Definition:** A construction of a composite register is *correct* iff all of its histories are atomic. □

This correctness condition, while intuitive, is rather difficult to use. We now present a lemma that gives a set of conditions that are sufficient for establishing that a history is atomic. Intuitively, a history is atomic if each operation in the history can be shrunk to a point; that is, there exists a point between the first and last events of each operation at which the operation appears to take effect. For this reason, the following lemma is referred to as the "Shrinking Lemma." The proof of this lemma is given in [2].

**Shrinking Lemma:** A history $h$ is atomic if for each $k$, where $0 \leq k < K$, there exists a function $\phi_k$ that maps every Read operation and $k$-Write operation in $h$ to some natural number, such that the following five conditions hold.

- *Uniqueness*: For each pair of distinct $k$-Write operations $v$ and $w$ in $h$, $\phi_k(v) \neq \phi_k(w)$. Furthermore, if $v$ precedes $w$, then $\phi_k(v) < \phi_k(w)$.
- *Integrity*: For each Read operation $r$ in $h$, and for each $k$ in the range $0 \leq k < K$, there exists a $k$-Write operation $w$ in $h$ such that $\phi_k(r) = \phi_k(w)$. Furthermore, the value Read by $r$ for component $k$ is the same as the value Written by $w$.
- *Proximity*: For each Read operation $r$ in $h$ and each $k$-Write operation $w$ in $h$, if $r$ precedes $w$ then $\phi_k(r) < \phi_k(w)$, and if $w$ precedes $r$ then $\phi_k(w) \leq \phi_k(r)$.
- *Read Precedence*: For each pair of Read operations $r$ and $s$ in $h$, if $(\exists k :: \phi_k(r) < \phi_k(s))$ or if $r$ precedes $s$, then $(\forall k :: \phi_k(r) \ \leq \ \phi_k(s))$.
- *Write Precedence*: For each Read operation $r$ in $h$, and each $j$-Write operation $v$ and $k$-Write operation $w$ in $h$, where $0 \leq j < K$ and $0 \leq k < K$, if $v$ precedes $w$ and $\phi_k(w) \leq \phi_k(r)$, then $\phi_j(v) \leq \phi_j(r)$. □

# 3 $K/L/M/N$ Construction

As stated in the introduction, we show in [2] that atomic registers can be used to implement a $K/L/1/N$ composite register. Therefore, to show that atomic registers can be used to implement a $K/L/M/N$ composite register, it suffices to construct a $K/L/M/N$ composite register from $K/L/1/N$ composite registers. In this section, we give such a construction.

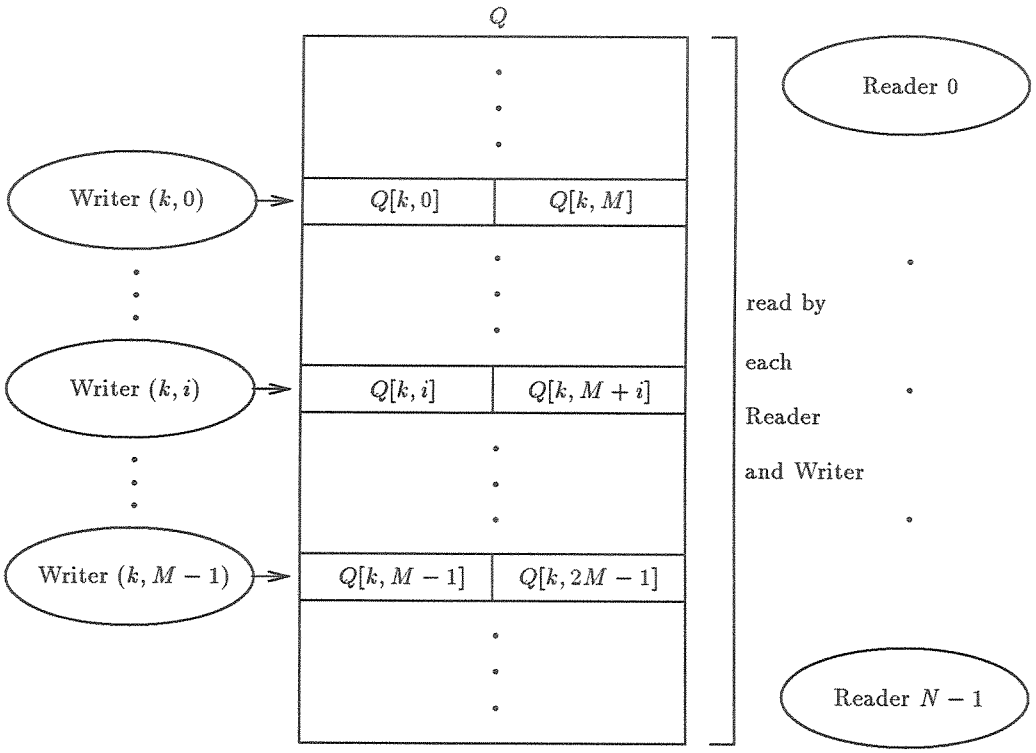


Figure 2: $K/L/M/N$ construction architecture.

An informal description of the construction is presented in Section 3.1, and the correctness proof is given in Section 3.2.

## 3.1 Informal Description

The architecture of our $K/L/M/N$ construction is shown in Figure 2. (This figure only depicts the $M$ Writers for component $k$ of the constructed register.) There is a single shared variable, namely $Q$. Variable $Q$ is a $KM/L'/1/KM + N$ composite register, where (as shown later) $L'$ is $O(L + M \log M)$. As seen in Figure 2, each pair of variables $Q[k, i]$ and $Q[k, M + i]$, where $0 \leq k < K$ and $0 \leq i < M$, corresponds to a single component of $Q$. Each of the $KM$ Writer programs writes to one component of $Q$, and all $KM + N$ programs read $Q$.

**Terminology:** In order to avoid confusion, we henceforth use the term "component" only when referring to the *constructed* composite register. We call each $Q[k, j]$, where $0 \leq k < K$ and $0 \leq j < 2M$, an "element" of $Q$. □

The type definition for $Q$, along with its initialization, is given in Figure 3. The fields of each element of variable $Q$ are as follows.

```
type Qtype = record   val : valtype;
                      tag : integer;   /* as shown in the appendix, a range of 0..8M − 4 suffices */
                      bid, eid : integer;   /* auxiliary variables */
                      wc : 0..2;
                      seq : array[0..2M − 1] of 0..2M;
                      flag : array[0..2M − 1] of boolean;
                      match : array[0..1] of boolean
              end
shared var
      Q : array[0..K − 1][0..2M − 1] of Qtype;
      P : array[0..K − 1] of integer   /* auxiliary variable */
initialization
      (∀k, m : 0 ≤ k < K ∧ 0 ≤ m < 2M : Q[k, m].wc = 2 ∧ Q[k, m].tag = 0 ∧
      ¬Q[k, m].match[0] ∧ ¬Q[k, m].match[1] ∧
      (∀n : 0 ≤ n < 2M : ¬Q[k, m].flag[n] ∧ Q[k, m].seq[n] = m))
```

Figure 3: Shared variable declarations for $K/L/M/N$ construction.

```
program Reader(j : 0..N − 1) returns array[0..K − 1] of valtype
private var
      x : Qtype;
      k : 0..K − 1;
      n : 0..2M − 1;
      max : array[0..K − 1] of 0..2M − 1;
      val : array[0..K − 1] of valtype;
      id : array[0..K − 1] of integer   /* auxiliary variable */
begin
  0:  read x := Q;
  1:  for k = 0 to K − 1 do
        select max[k] such that SA(x, k, max[k]) ∧
                    (∀n : SA(x, k, n) : (x[k, n].tag, n) ≤ (x[k, max[k]].tag, max[k]));
        val[k], id[k] := x[k, max[k]].val, x[k, max[k]].eid
      od;
  2:  return(val[0], . . . , val[K − 1])
end
```

Figure 4: Reader program for $K/L/M/N$ construction.

```
program Writer(k : 0..K − 1; m : 0..M − 1; val : valtype)
private var
        x, y, z : Qtype;
        seq : array[0..2M − 1] of 0..2M;
        flag : array[0..2M − 1] of boolean;
        match : array[0..1] of boolean;
        i, j, max, n : 0..2M − 1;
        bid, eid : integer;   /* auxiliary variables */
        tag : integer    /* as shown in the appendix, a range of 0..8M − 4 suffices */
begin
    0:  ⟨if i = m then i, j := M + m, m else i, j := m, M + m fi; bid, P[k] := P[k], P[k] + 1⟩;

        /* compute sequence numbers and flags */
    1:  read x := Q;
    2:  select seq[i] such that (∀n : 0 ≤ n < 2M : seq[i] ≠ x[k, n].seq[i]);
    3:  tag, eid, match[0..1], flag[i] := x[k, i].tag, x[k, i].eid, x[k, i].match[0..1], false;
    4:  for n = 0 to 2M − 1 skip i do
            seq[n], flag[n] := x[k, n].seq[n], (x[k, i].seq[n] = x[k, j].seq[n] = x[k, n].seq[n])
        od;
    5:  write Q[k, i] := (val, tag, bid, eid, 0, seq[0..2M − 1], flag[0..2M − 1], match[0..1]);

        /* compute tag and match[0] */
    6:  read y := Q;
    7:  select max such that WA(y, k, max) ∧
                (∀n : WA(y, k, n) : (y[k, n].tag, n) ≤ (y[k, max].tag, max));
    8:  tag := y[k, max].tag + 1;
    9:  match[0] := (∃n : 0 ≤ n < 2M  ∧  n ≠ i : y[k, n].flag[i]  ∧  y[k, n].seq[i] = seq[i]);
   10:  write Q[k, i] := (val, tag, bid, eid, 1, seq[0..2M − 1], flag[0..2M − 1], match[0..1]);

        /* compute match[1] */
   11:  read z := Q;
   12:  match[1] := (∃n : 0 ≤ n < 2M  ∧  n ≠ i : z[k, n].flag[i]  ∧  z[k, n].seq[i] = seq[i]);
   13:  ⟨eid, P[k] := P[k], P[k] + 1;
        write Q[k, i] := (val, tag, bid, eid, 2, seq[0..2M − 1], flag[0..2M − 1], match[0..1])⟩
end
```

Figure 5: Writer program for $K/L/M/N$ construction.

*val*: A value Written by a Write operation.

*tag*: An integer "tag" that is appended to a value. As explained below, each *tag* field can be restricted to range over $0..8M-4$.

*bid*: An integer auxiliary variable updated at the "beginning" of a Write operation.

*eid*: An integer auxiliary variable updated at the "end" of a Write operation.

*wc*: A modulo-3 integer "write counter" that is incremented each time a Write operation writes to a particular element of $Q$.

The remaining fields are used to bound the size of the *tag* fields. This is described below.

*seq*: An array of $2M$ "sequence numbers." Each sequence number is a modulo-$(2M+1)$ integer.

*flag*: An array of $2M$ bits.

*match*: A pair of bits.

The Reader and Writer programs are shown in Figures 4 and 5, respectively. In the Reader and Writer programs, we use a special syntax in order to distinguish reads and writes of shared variables from reads and writes of private variables. A program reads a given shared variable $V$ by executing a statement of the form "**read** $x := V$," where $x$ is a private variable of the same type as $V$. A program writes a shared variable $V$ by executing a statement of the form "**write** $V := x$." Notice in Figure 5 that the private auxiliary variable *bid* of each $k$-Write operation is updated by atomically incrementing the shared auxiliary variable $P[k]$ in line 0, and that the private auxiliary variable *eid* is updated by atomically incrementing $P[k]$ in line 13. This is denoted by the angle brackets '$\langle$' and '$\rangle$'.

Each Write operation writes to a particular element of $Q$; we call a Write operation that writes to $Q[k,j]$ a $(k,j)$-*Write operation*. The Write operations for a given component follow a protocol that is similar to that used in the multiple-Writer atomic register construction of Vitanyi and Awerbuch [15]. Each Write operation for a particular component appends a "tag" to the value that it Writes; a Write operation computes its tag by incrementing the value of the maximum tag that it reads from the elements of $Q$ corresponding to its component. A Read operation returns the value from the element of that component with the maximum tag (ties are broken using the indices of the Writers). The $K$ values returned by a Read operation constitute a consistent snapshot since all of the elements of variable $Q$ are read in a single statement.

As seen in Figure 5, each Write operation writes to a particular element of $Q$ three times. As we shall see later, the value of a particular component is determined only by the elements of that component that are up-to-date, i.e., that were last written by a completed Write operation. Thus, the value of a particular component is well-defined at some state only if

there exists an element of that component that is up-to-date at that state. To ensure that this is always the case, successive operations of the same Writer write to different elements of $Q$. In particular, the operations of Writer $(k, m)$, where $0 \leq k < K$ and $0 \leq m < M$, alternate between writing to $Q[k, m]$ and $Q[k, m + M]$. (This is why each component of the constructed register consists of $2M$ elements of $Q$, instead of just $M$.) This strategy guarantees that at least half of the elements of a component are up-to-date at every state.

In order to determine which element of a component has the maximum tag, it is necessary only to consider those elements that have been written "recently." By exploiting this fact, it is possible to bound the size of the *tag* fields. We say that a recently written element is "alive." The alive elements are identified by including a number of additional fields in each element of $Q$. One of these fields is an array $seq[0..2M-1]$ of "sequence numbers." The field $Q[k, j].seq[j]$ is the "primary" sequence number of the most recent $(k, j)$-Write operation. The field $Q[k, j].seq[n]$, where $n \neq j$, is a copy of the primary sequence number of the most recent $(k, n)$-Write operation as read from $Q[k, n].seq[n]$ by the most recent $(k, j)$-Write operation.

A $(k, j)$-Write operation seeks to make the value of its primary sequence number, $Q[k, j].seq[j]$, distinct from all copies of it. At the same time, it updates its copy of each other primary sequence number to correspond to the current value of that sequence number; more specifically, it tries to make the value of $Q[k, j].seq[n]$ equal to that of $Q[k, n].seq[n]$, for each $n \neq j$. If the primary sequence number for a given element remains unchanged for a sufficiently long period of time, then it will eventually be copied by some Write operation. An element is no longer alive once its primary sequence number has been copied by three successive operations of some Writer. (Note that if $v$, $v'$, and $v''$ are three successive operations of the same Writer, then $v$ and $v''$ are $(k, i)$-Write operations and $v'$ is a $(k, i + M \textbf{ modulo } 2M)$-Write operation for some $i$ in the range $0 \leq i < 2M$.)

The above condition is detected by means of the *flag* and *match* fields. The bit $Q[k, n].flag[j]$, where $j \neq n$, is set by a $(k, n)$-Write operation if three successive operations of its Writer have read the same value from $Q[k, j].seq[j]$. The bits $Q[k, j].match[0]$ and $Q[k, j].match[1]$ are set by a $(k, j)$-Write operation if it detects that its own primary sequence number, $Q[k, j].seq[j]$, has been read by three successive operations of some other Writer.

As shown in an appendix, the tags of the alive elements of a given component are within a range of size $4M-1$. Therefore, we can restrict the size of each tag to range over $0..8M-4$. The maximum tag for the alive elements is then determined with respect to this range.

The Readers use a slightly stronger "aliveness" condition than that of the Writers. According to the Readers' aliveness condition, if an element is alive, then its *wc* field equals 2. This ensures that a Read operation Reads a value as Written by a completed Write operation. Thus, the value of a particular component is determined only by the up-to-date elements of that component. According to the Writers' aliveness condition, if an element is alive, then its *wc* field equals either 1 or 2. This ensures that a Write operation chooses its

tag based upon the most recently written tags. In the Reader program, the alive elements are determined by the predicate $SA(Q,k,j)$, and in the Writer program, the alive elements are determined by the predicate $WA(Q,k,j)$; $SA$ stands for "strongly alive" and $WA$ stands for "weakly alive." These predicates are defined as follows.

$$\begin{aligned} WA(Q,k,j) \equiv\ & Q[k,j].wc \geq 1 \ \wedge\ \neg Q[k,j].match[0] \ \wedge \\ & (\forall n : n \neq j \ \wedge\ Q[k,n].wc = 2 : Q[k,n].flag[j] \ \Rightarrow \\ & Q[k,n].seq[j] \neq Q[k,j].seq[j]) \end{aligned}$$

$$SA(Q,k,j) \equiv\ \ WA(Q,k,j) \ \wedge\ Q[k,j].wc = 2 \ \wedge\ \neg Q[k,j].match[1]$$

According to the first conjunct in the definition of $WA$, $WA(Q,k,j)$ fails to hold if the last Write operation to write to $Q[k,j]$ has done so only once. According to the second conjunct, $WA(Q,k,j)$ fails to hold if the last Write operation to update the bit $Q[k,j].match[0]$ detected that its primary sequence number, $Q[k,j].seq[j]$, was read by at least three successive operations of some other Writer. According to the third conjunct, $WA(Q,k,j)$ fails to hold if at least three successive operations of some Writer have read the current value of $Q[k,j].seq[j]$.

According to the first conjunct in the definition of $SA$, $SA(Q,k,j)$ fails to hold if $WA(Q,k,j)$ fails to hold. According to the second conjunct, $SA(Q,k,j)$ fails to hold if the last Write operation to write to $Q[k,j]$ has not completed. According to the third conjunct, $SA(Q,k,j)$ fails to hold if the last Write operation to update the bit $Q[k,j].match[1]$ detected that its primary sequence number, $Q[k,j].seq[j]$, was read by at least three successive operations of some other Writer.

We now compute the space complexity of our $K/L/M/N$ construction by determining the number of shared $1/1/1/1$ composite registers used in the construction. Let $B(K,L,M,N)$ denote the number of shared $1/1/1/1$ composite registers required to construct a $K/L/M/N$ composite register. If we remove the auxiliary *bid* and *eid* fields from our $K/L/M/N$ construction, then the complexity of the fields of each component of variable $Q$ is as follows.

- *val* uses $L$ bits.
- *tag* uses $\log(8M-3)$ bits.
- *wc* uses 2 bits.
- $seq[i]$, $0 \leq i < 2M$, uses $\log(2M+1)$ bits.
- $flag[i]$, $0 \leq i < 2M$, uses 1 bit.
- $match[i]$, $0 \leq i \leq 1$, uses 1 bit.

Thus, variable $Q$ is a $KM/L'/1/KM+N$ composite register, where $L' = O(L + M \log M)$. Hence, the space complexity of our $K/L/M/N$ construction is $B(K,L,M,N) = B(KM,L',$

$1, KM+N)$. By using the value of $B(K, L, 1, N)$ as computed in [2], we have $B(K, L, 1, N) = O(KN^2 + K^2LN + K^3L)$. Therefore, we get

$$B(K, L, M, N) = O(K^3LM^3 + K^3M^4 \log M + K^2LM^2N + K^2M^3N \log M + KMN^2) \ .$$

We compute the time complexity by determining the number of reads and writes of shared $1/L/1/N$ composite registers (i.e., single-writer atomic registers) required to Read and Write the constructed register (for simplicity, we do not go down to the level of $1/1/1/1$ registers when computing the time complexity). Let $TR(K, L, M, N)$ and $TW(K, L, M, N)$ denote the time complexity for Reading and Writing, respectively, a $K/L/M/N$ composite register. The time complexity of a Read operation is

$$TR(K, L, M, N) = TR(KM, L', 1, KM + N) \ .$$

The time complexity of a Write operation is

$$TW(K, L, M, N) = 3TR(KM, L', 1, KM + N) + 3TW(KM, L', 1, KM + N) \ .$$

By using the values of $TR(K, L, 1, N)$ and $TW(K, L, 1, N)$ as computed in [2], we have $TR(K, L, 1, N) = O(2^K)$ and $TW(K, L, 1, N) = O(N + 2^K)$. Therefore, if variable $Q$ is implemented using the $K/L/1/N$ construction given in [2], then the time complexity of a Read would be $O(2^{KM})$ and the time complexity of a Write would be $O(N + 2^{KM})$. If, on the other hand, a polynomial-time $K/L/1/N$ construction is used to implement $Q$, then the time complexity for our $K/L/M/N$ construction would also be polynomial.

## 3.2 Correctness Proof

In this section, we prove that the construction is correct as is, i.e., with unbounded tags. In an appendix, we show how to transform the original algorithm into one with bounded tags.

The correctness proof is based on the Shrinking Lemma. We first define functions $\phi_0, \ldots, \phi_{K-1}$ for a given history, and then show that the defined $\phi$'s satisfy the five conditions of Uniqueness, Integrity, Proximity, Read Precedence, and Write Precedence. The following notations and definitions are used in the rest of the paper.

**Notation:** Unless stated otherwise, we assume that $k$ ranges over $\{0, \ldots, K-1\}$, and that $i$, $j$, and $n$ each range over $\{0, \ldots, 2M-1\}$. We use $p$ and $q$ to denote arbitrary operations, $r$ and $s$ to denote Read operations, $v$ and $w$ to denote Write operations, and $t$ and $u$ to denote states. Unless stated otherwise, we assume that $v$ and $w$ are $k$-Write operations. Also, we use $\oplus$ to denote modulo-$2M$ addition. □

**Notation:** In order to avoid using too many parentheses, we define a binding order for the symbols that we use. The following is a list of these symbols, grouped by binding power; the groups are ordered from highest binding power to lowest.

$[\,], (\,), |\,|$
.
!
$\neg$, :
$+, -, \oplus$
$=, \neq, <, >, \leq, \geq, \prec, \preceq, \in$
$\wedge, \vee$
$\Rightarrow, \equiv$
$\models$ □

**Definition:** If event $e$ precedes event $f$, then we write $e \prec f$. We let $(e \preceq f) \equiv (e = f \vee e \prec f)$. □

**Definition:** Let $p$ be an operation, and let $x$ be any private variable of $p$. Then, $p!x$ denotes the final value of variable $x$ as assigned by operation $p$. □

**Definition:** Let $p$ be an operation of some Reader or Writer program and let $i$ be a label of a statement in that program. We denote the event corresponding to the execution of statement $i$ in operation $p$ by $p{:}i$. □

**Definition:** If $E$ is an expression that holds at state $t$, then we write $t \models E$. □

Whenever we say that a given assertion holds without referring to a particular state, we mean that the assertion is an *invariant*; i.e., it is true at each state of every history.

**Assumption:** We assume that each state in every history is distinct. This assumption is easy to ensure by introducing an integer auxiliary variable that is incremented with each event. □

**Definition:** Consider the history $t_0 \xrightarrow{e_0} \cdots t_i \xrightarrow{e_i} t_{i+1} \cdots$. We say that $t_i$ is the state *prior to* the event $e_i$ and $t_{i+1}$ is the state *following* $e_i$. Similarly, $e_i$ is the event *prior to* the state $t_{i+1}$ and the event *following* state $t_i$. □

Note that the events prior to and following a given state are uniquely defined since, by assumption, each state appears at most once in a given history.

**Definition:** Let $e$ be the event corresponding to the execution of the statement **read** $x := Y$ in an operation $p$, where $x$ is a private variable and $Y$ is a shared variable. If $f$ is the last event to write $Y$ before $e$, then we say that $f$ *determines* $p!x$. □

**Definition:** If $p$ and $q$ are successive operations of the same Reader or Writer program,

then we write $p = pred(q)$ and $q = succ(p)$. □

**Definition:** An assertion $A$ is *stable* iff for every pair of consecutive states in any history, if $A$ holds in the first state, then it also holds in the second state. □

**Definition:** Let $X$ be a shared variable of the construction, and let $p$ be an operation. The assertion $last(X) = p$ holds at a state iff the last event to write to $X$ before that state is an event of $p$. □

**Definition:** Let $e$ be an event in some history. Then, $after(e)$ is true at a state iff the state occurs after the event $e$. □

Based on the definitions of $WA$ and $SA$ given in Section 3.1, we define four predicates: *walive* indicates whether a Write operation is "weakly alive," *salive* indicates whether a Write operation is "strongly alive," *wpref* indicates whether a weakly alive operation is "preferable," i.e., has the largest tag for its component, and *spref* indicates whether a strongly alive Write operation is preferable.

**Definition:** Let $w$ be a $(k, j)$-Write operation. Then,

$$\begin{aligned}
walive(w,k) &\equiv last(Q[k,j]) = w \ \wedge \ WA(Q,k,j) \\
salive(w,k) &\equiv last(Q[k,j]) = w \ \wedge \ SA(Q,k,j) \\
wpref(w,k) &\equiv walive(w,k) \ \wedge \ (\forall v : walive(v,k) : (v!tag, v!i) \leq (w!tag, j)) \\
spref(w,k) &\equiv salive(w,k) \ \wedge \ (\forall v : salive(v,k) : (v!tag, v!i) \leq (w!tag, j))
\end{aligned}$$ □

As mentioned in Section 3.1, each element of $Q$ includes two auxiliary fields *bid* and *eid*. These variables have been introduced in order to facilitate the definition of $\phi_0, \ldots, \phi_{K-1}$.

**Definition:** Let $r$ be a Read operation and let $w$ be a $k$-Write operation. Then, $\phi_k$ is defined as follows.

$$\begin{aligned}
\phi_k(r) &\equiv r!id[k] \\
\phi_k(w) &\equiv \begin{cases} w!eid & \text{if } spref(w,k) \text{ holds at the state following } w : 13 \\ w!bid & \text{otherwise} \end{cases}
\end{aligned}$$

□

Before establishing the conditions of Uniqueness, Integrity, Proximity, Read Precedence, and Write Precedence, we first prove a number of lemmas. The following lemma gives us a means for determining the value of $Q[k, j]$ at a given state.

**Lemma 1:** Let $v$ be a $(k, j)$-Write operation. Then,

- $last(Q[k,j]) = v \ \Rightarrow \ Q[k,j].val = v!val \ \wedge \ Q[k,j].bid = v!bid \ \wedge \ (\forall n :: Q[k,j].seq[n] = v!seq[n] \ \wedge \ Q[k,j].flag[n] = v!flag[n])$

- $last(Q[k,j]) = v \ \wedge \ after(v{:}10) \ \Rightarrow \ Q[k,j].wc \geq 1 \ \wedge \ Q[k,j].tag = v!tag \ \wedge \ Q[k,j].match[0] = v!match[0]$
- $last(Q[k,j]) = v \wedge after(v{:}13) \Rightarrow Q[k,j].wc = 2 \wedge Q[k,j].match[1] = v!match[1] \wedge Q[k,j].eid = v!eid$

**Proof:** The lemma holds because $v{:}5$ assigns the values $v!val$, $v!bid$, 0, $v!seq[0..2M-1]$, and $v!flag[0..2M-1]$ to the fields $val$, $bid$, $wc$, $seq[0..2M-1]$, and $flag[0..2M-1]$, respectively, of $Q[k,j]$, while leaving the value of each other field unchanged; $v{:}10$ assigns the values $v!tag$, 1, and $v!match[0]$ to the fields $tag$, $wc$, and $match[0]$, respectively, of $Q[k,j]$, while leaving the value of each other field unchanged; and $v{:}13$ assigns the values $v!eid$, 2, and $v!match[1]$ to the fields $eid$, $wc$, and $match[1]$, respectively, of $Q[k,j]$, while leaving the value of each other field unchanged. □

According to the next lemma, if Write operation $v$ is "strongly alive" then it is also "weakly alive."

**Lemma 2:** $(\forall v, k :: salive(v,k) \ \Rightarrow \ walive(v,k))$.

**Proof:** Follows by the definition of $salive$, $walive$, $SA$, and $WA$. □

According to the next lemma, the initial $(k,j)$-Write operation assigns the value false to $Q[k,j].flag[n]$ for each $n \neq j$.

**Lemma 3:** Let $v$ be the initial $(k,j)$-Write operation, and let $n \neq j$. Then, $v!flag[n]$ is false.

**Proof:** Let $v$ be as defined in the lemma, and let $n \neq j$. We show that $v!flag[n]$ is false. Let $t$ be the state prior to the event $v{:}1$. By the program for the Writer, $t \ \models \ Q = v!x$. Also, by the program for the Writer,

$$v!flag[n] = (v!x[k,j].seq[n] = v!x[k,j'].seq[n] = v!x[k,n].seq[n]),$$

where $j' = j \oplus M$. Therefore, we can meet our proof obligation by showing that $t \ \models \ Q[k,j].seq[n] \neq Q[k,n].seq[n]$.

Because $v$ is the initial $(k,j)$-Write operation, by the definition of the initial state, $t \ \models \ Q[k,j].seq[n] = j$. If $t \ \models \ Q[k,n].seq[n] = n$, then clearly $t \ \models \ Q[k,j].seq[n] \neq Q[k,n].seq[n]$. So, assume that $t \ \models \ Q[k,n].seq[n] \neq n$. In this case, the value of $Q[k,n].seq[n]$ at state $t$ differs from its initial value. Therefore, there exists a $(k,n)$-Write operation $w$ such that $t \ \models \ last(Q[k,n]) = w \ \wedge \ Q[k,n].seq[n] = w!seq[n]$. Because $last(Q[k,n]) = w$ at state $t$, i.e., the state prior to $v{:}1$, by the program for the Writer $w{:}5 \prec v{:}1$. This implies that $w{:}1 \prec v{:}1$. Hence, because $v$ is the initial $(k,j)$-Write operation, by the definition of the initial state, $Q[k,j].seq[n] = j$ at the state prior to $w{:}1$. This

implies that $w!x[k,j].seq[n] = j$. By the program for the Writer, $w!seq[n] \neq w!x[k,j].seq[n]$. Therefore, $w!seq[n] \neq j$. Hence, $t \models Q[k,j].seq[n] \neq Q[k,n].seq[n]$. □

In the next lemma, we consider the case in which three successive operations of one Writer have copied the current sequence number of another Writer.

**Lemma 4:** Suppose that $t \models last(Q[k,n]) = v \wedge last(Q[k,j]) = w$, where $n \neq j$. If $v!flag[j] \wedge v!seq[j] = w!seq[j]$, then there exists $v'$ such that $v' = pred(v) \wedge w{:}5 \prec v'{:}1 \wedge v'!seq[j] = w!seq[j]$.

**Proof:** Let $v$, $w$, and $t$ be as defined in the lemma. Suppose that

$$v!flag[j] \wedge v!seq[j] = w!seq[j] \quad . \tag{1}$$

Our proof obligation is to show that there exists $v'$ such that $v' = pred(v) \wedge w{:}5 \prec v'{:}1 \wedge v'!seq[j] = w!seq[j]$.

We first show that $w{:}1 \prec v{:}5$. Assume, to the contrary, that $v{:}5 \prec w{:}1$. By the program for the Writer, $w{:}1 \prec w{:}5$. Let $e$ be the event prior to $t$. Because $t \models last(Q[k,j]) = w$, we have $w{:}5 \preceq e$. Therefore,

$$v{:}5 \prec w{:}1 \prec w{:}5 \preceq e \quad .$$

Because $last(Q[k,n]) = v$ at state $t$ (the state following $e$), the above precedence assertion implies that $last(Q[k,n]) = v$ at every state between $v{:}5$ and $e$. In particular, $last(Q[k,n]) = v$ at the state prior to $w{:}1$. Therefore, by Lemma 1, $Q[k,n].seq[j] = v!seq[j]$ at the state prior to $w{:}1$. By the program for the Writer, this implies that $w!x[k,n].seq[j] = v!seq[j]$. Furthermore, because $w$ is a $(k,j)$-Write operation, from the program for the Writer, $w!seq[j] \neq w!x[k,n].seq[j]$. Therefore, $w!seq[j] \neq v!seq[j]$, which contradicts (1). Thus, our assumption that $v{:}5 \prec w{:}1$ is false, i.e., $w{:}1 \prec v{:}5$.

Because $v!flag[j]$ holds, by Lemma 3, $v$ is not the initial $(k,n)$-Write operation. Therefore, there exist Write operations $v'$ and $v''$ such that $v' = pred(v)$ and $v'' = pred(v')$. Note that $v'$ is a $(k, n \oplus M)$-Write operation and $v''$ is a $(k,n)$-Write operation. Because $v!flag[j]$ is true, by the program for the Writer, $v''!seq[j] = v'!seq[j] = v!seq[j]$. Therefore, by (1),

$$v''!seq[j] = v'!seq[j] = w!seq[j] \quad . \tag{2}$$

We now show that $w{:}1 \prec v''{:}5$. Assume, to the contrary, that $v''{:}5 \prec w{:}1$. Then, because $w{:}1 \prec v{:}5$, we have $v''{:}5 \prec w{:}1 \prec v{:}5$. Since $v''$ and $v$ are consecutive $(k,n)$-Write operations, this implies that $last(Q[k,n]) = v''$ at the state prior to $w{:}1$. Therefore, by Lemma 1, $Q[k,n].seq[j] = v''!seq[j]$ at the state prior to $w{:}1$. Hence, by the program for the Writer, $w!x[k,n].seq[j] = v''!seq[j]$. Thus, because $w!seq[j] \neq w!x[k,n].seq[j]$, we conclude that $w!seq[j] \neq v''!seq[j]$. However, this contradicts (2). Thus, our assumption that $v''{:}5 \prec w{:}1$ is false, i.e., $w{:}1 \prec v''{:}5$.

Thus far we have established that there exists $v'$, where $v' = pred(v)$, such that $v'!seq[j] = w!seq[j]$. Our remaining proof obligation is to show that $w{:}5 \prec v'{:}1$. Assume, to the contrary, that $v'{:}1 \prec w{:}5$. As established above, $w{:}1 \prec v''{:}5$. Because $v'' = pred(v')$, $v''{:}5 \prec v'{:}1$. Thus,

$$w{:}1 \prec v'{:}1 \prec w{:}5 \ . \tag{3}$$

Let $n' = n \oplus M$. Then, $v'$ is a $(k, n')$-Write operation. Because $w$ is a $(k, j)$-Write operation (and because the $(k, j)$-Write operations are totally ordered), (3) implies that $n' \neq j$.

Because $w$ is a $(k, j)$-Write operation, by (3) and the program for the Writer, $Q[k, j].seq[j]$ has the same value both at the state prior to $w{:}1$ and the state prior to $v'{:}1$. Therefore, by the program for the Writer, $w!x[k, j].seq[j] = v'!x[k, j].seq[j]$. Because $w$ is a $(k, j)$-Write operation, i.e., $w!i = j$, we have $w!seq[j] \neq w!x[k, j].seq[j]$. Because $v'$ is not a $(k, j)$-Write operation, i.e., $v'!i = n' \neq j$, we have $v'!seq[j] = v'!x[k, j].seq[j]$. Therefore, $w!seq[j] \neq v'!seq[j]$. However, this contradicts (2). Thus, our assumption that $v'{:}1 \prec w{:}5$ is false, i.e., $w{:}5 \prec v'{:}1$. □

According to the following lemma, if a completed $k$-Write operation $w$ is not "strongly alive" then there exists another completed $k$-Write operation $v$ such that $w{:}5 \prec v{:}1$.

**Lemma 5:** $(after(w{:}13) \wedge \neg salive(w, k)) \Rightarrow (\exists v :: w{:}5 \prec v{:}1 \ \wedge \ after(v{:}13))$.

**Proof:** Suppose that $after(w{:}13) \ \wedge \ \neg salive(w, k)$ holds at some state $t$, where $w$ is a $(k, j)$-Write operation. Our proof obligation is to show that there exists a $k$-Write operation $v$ such that $w{:}5 \prec v{:}1$ and $t \models after(v{:}13)$.

We first dispose of the case in which $t \models last(Q[k, j]) \neq w$. In this case, because $after(w{:}13)$ holds at $t$, there exists a $(k, j)$-Write operation $w'$, where $w$ precedes $w'$, such that $t \models last(Q[k, j]) = w'$. Because successive operations of the same Writer write to different elements of $Q$, this implies that there exists a $(k, j \oplus M)$-Write operation $v$ such that $v = succ(w)$ and $after(v{:}13)$ holds at $t$. Because $v = succ(w)$, we have $w{:}5 \prec v{:}1$. This establishes our proof obligation.

In the remainder of the proof, we assume that $t \models last(Q[k, j]) = w$. In this case, because $t \models \neg salive(w, k)$, by the definition of $salive$, $t \models \neg SA(Q, k, j)$. We now show that there exists a state $u$, where $u$ either equals or occurs before $t$, such that for some $n \neq j$ the following expression holds.

$$u \models last(Q[k, j]) = w \ \wedge \ Q[k, n].flag[j] \ \wedge \ Q[k, n].seq[j] = Q[k, j].seq[j] \tag{4}$$

Because $t \models last(Q[k, j]) = w \ \wedge \ after(w{:}13)$, by Lemma 1, $t \models Q[k, j].wc = 2$. Therefore, since $t \models \neg SA(Q, k, j)$, by the definition of $SA$, there are two possibilities to consider:

(i) there exists $n \neq j$ such that $t \models Q[k, n].flag[j] \ \wedge \ Q[k, n].seq[j] = Q[k, j].seq[j]$, or

(ii) $t \models Q[k,j].match[0] \vee Q[k,j].match[1]$.

If (i) holds, then take $u = t$. Then, by assumption $u \models last(Q[k,j]) = w$, which establishes (4).

Now, suppose that (ii) holds, i.e., $Q[k,j].match[c]$ holds at $t$, where $c$ equals either 0 or 1. Because $t \models last(Q[k,j]) = w \wedge after(w{:}13)$, by Lemma 1, $t \models Q[k,j].match[c] = w!match[c]$. Therefore, $w!match[c]$ is true. If $c = 0$, then let $u$ be the state prior to $w{:}6$, and if $c = 1$, then let $u$ be the state prior to $w{:}11$. In either case, because $after(w{:}13)$ holds at $t$, $u$ occurs before $t$. Also, in either case, $u \models last(Q[k,j]) = w$. We establish that (4) holds for the case in which $c = 0$; the case in which $c = 1$ is similar. Because $w!match[0]$ holds, there exists $n \neq j$ such that $w!y[k,n].flag[j] \wedge w!y[k,n].seq[j] = w!seq[j]$. By the definition of state $u$, $u \models Q[k,n] = w!y[k,n] \wedge Q[k,j].seq[j] = w!seq[j]$. Therefore, $u \models Q[k,n].flag[j] \wedge Q[k,n].seq[j] = Q[k,j].seq[j]$. This establishes (4).

We now use (4) to establish our proof obligation. Let $v$ be the Write operation such that $u \models last(Q[k,n]) = v$. ($v$ exists because $Q[k,n].flag[j]$ is initially false.) Because $u \models last(Q[k,n]) = v \wedge last(Q[k,j]) = w$, by Lemma 1, $u \models Q[k,n].flag[j] = v!flag[j] \wedge Q[k,n].seq[j] = v!seq[j] \wedge Q[k,j].seq[j] = w!seq[j]$. Therefore, by (4), $v!flag[j] \wedge v!seq[j] = w!seq[j]$. Therefore, by Lemma 4, there exists $v'$ such that $v' = pred(v)$ and $w{:}5 \prec v'{:}1$. Because $v'$ precedes $v$ and $last(Q[k,n]) = v$ at state $u$, $after(v'{:}13)$ holds at state $u$. By the definition of $u$, $u$ either equals or occurs before $t$. Thus, by the definition of *after*, $after(v'{:}13)$ holds at state $t$. This establishes our proof obligation. □

The next lemma shows that $salive(w,k)$ holds for some $w$ at every state that occurs after the initial $k$-Write operation.

**Lemma 6:** $(\exists v :: after(v{:}13)) \Rightarrow (\exists w :: salive(w,k))$.

**Proof:** Let $t$ be a state such that for some $k$-Write operation $v$, $t \models after(v{:}13)$. We show that there exists some $k$-Write operation $w$ such that $t \models salive(w,k)$. Let $S$ denote the set of $k$-Write operations defined as follows: $p \in S$ iff $t \models after(p{:}13)$. By assumption, $S$ is nonempty. Let $w$ denote the Write operation in $S$ such that for each other $k$-Write operation $p$ in $S$, $p{:}5 \prec w{:}5$. Then, by Lemma 5, $t \models salive(w,k)$. □

As we now show, Lemma 6 implies that $(\exists j :: SA(Q,k,j))$ holds at every state. By the definition of $SA$, this implies that $(\exists j :: WA(Q,k,j))$ also holds at every state. As a result, the computation of $max[k]$ and $max$ in the Reader and Writer programs, respectively, is well-defined.

Let $v$ be the initial $k$-Write operation. Assume that $v$ is a $(k,j)$-Write operation. By our assumption concerning the initial Writes, $v$ precedes all other $k$-Write operations. Consider a state $t$. If $t$ occurs after the event $v{:}13$, then by Lemma 6, $t \models (\exists w :: salive(w,k))$. Therefore, by the definition of *salive*, $t \models (\exists n :: SA(Q,k,n))$.

Now, suppose that $t$ occurs before the event $v{:}13$. If $t$ occurs before $v{:}5$, then by the definition of the initial state, $t \models (\forall n' :: \neg Q[k,j].flag[n'])$. If $t$ occurs between $v{:}5$ and $v{:}13$, then $t \models Q[k,j].wc \neq 2$. Therefore,

$$t \models Q[k,j].wc \neq 2 \ \vee \ (\forall n' :: \neg Q[k,j].flag[n']) \ .$$

Because $v$ precedes all other $k$-Write operations, by the definition of the initial state,

$$\begin{aligned} t \ \models \ & (\forall n : n \neq j : Q[k,n].wc = 2 \ \wedge \ \neg Q[k,n].match[0] \ \wedge \\ & \neg Q[k,n].match[1] \ \wedge \ (\forall n' :: \neg Q[k,n].flag[n'])) \ . \end{aligned}$$

By the definition of $SA$, the above two assertions imply that $t \models (\forall n : n \neq j : SA(Q,k,n))$. Because, by assumption, $n$ ranges over $0 \leq n < 2M$, the range in this expression is not empty. Therefore, $t \models (\exists n :: SA(Q,k,n))$.

According to the next lemma, if a Write operation that has written to $Q$ at least twice is not "weakly alive" at some state, then it is forever after not "weakly alive."

**Lemma 7:** $\neg walive(w,k) \ \wedge \ after(w{:}10)$ is stable.

**Proof:** Let $t$ and $u$ be consecutive states such that $t \models \neg walive(w,k) \ \wedge \ after(w{:}10)$. By the definition of *after*, $u \models after(w{:}10)$. This reduces our proof obligation to showing that $u \models \neg walive(w,k)$.

Assume that $w$ is a $(k,j)$-Write operation. By the definition of *walive*, if $u \models last(Q[k,j]) \neq w$, then $u \models \neg walive(w,k)$. So, in this case, our proof obligation is satisfied. In the remainder of the proof, assume that $u \models last(Q[k,j]) = w$. Because $after(w{:}10)$ holds at state $u$, we have

$$u \models last(Q[k,j]) = w \ \wedge \ after(w{:}10) \ . \tag{5}$$

By the definition of *walive*, our proof obligation is to show that $WA(Q,k,j)$ is false at state $u$.

Because $t$ and $u$ are consecutive states and $t \models after(w{:}10)$, by (5) and the program for the Writer,

$$t \models last(Q[k,j]) = w \ \wedge \ after(w{:}10) \ . \tag{6}$$

Because $t \models last(Q[k,j]) = w \ \wedge \ \neg walive(w,k)$, by the definition of *walive*, $t \models \neg WA(Q,k,j)$. By (6) and Lemma 1, $t \models Q[k,j].wc \geq 1$. Therefore, by the definition of $WA$, $t \models Q[k,j].match[0]$ or there exists $n$, where $n \neq j$, such that $t \models Q[k,n].wc = 2 \ \wedge \ Q[k,n].flag[j] \ \wedge \ Q[k,n].seq[j] = Q[k,j].seq[j]$. We first dispose of the case $t \models Q[k,j].match[0]$. By Lemma 1 and (6), $t \models Q[k,j].match[0] = w!match[0]$. Thus, $w!match[0]$ is true. By Lemma 1 and (5), $u \models Q[k,j].match[0] = w!match[0]$. Hence, $u \models Q[k,j].match[0]$. Therefore, by the definition of $WA$, $u \models \neg WA(Q,k,j)$, which establishes our proof obligation.

In the remainder of the proof, we consider the other case, i.e., there exists $n$, where $n \neq j$, such that

$$t \models Q[k,n].wc = 2 \wedge Q[k,n].flag[j] \wedge Q[k,n].seq[j] = Q[k,j].seq[j] \quad . \tag{7}$$

Let $v$ be the $(k,n)$-Write operation such that

$$t \models last(Q[k,n]) = v \quad . \tag{8}$$

($v$ exists because $Q[k,n].flag[j]$ is initially false.) By (6) and (8) and Lemma 1, $t \models Q[k,n].flag[j] = v!flag[j] \wedge Q[k,n].seq[j] = v!seq[j] \wedge Q[k,j].seq[j] = w!seq[j]$. Hence, by (7),

$$v!flag[j] \wedge v!seq[j] = w!seq[j] \quad . \tag{9}$$

We now consider two cases, depending on whether $u \models last(Q[k,n]) = v$.

Case 1: $u \models last(Q[k,n]) = v$. By (7) and (8), $t \models last(Q[k,n]) = v \wedge Q[k,n].wc = 2$. By the program for the Writer this implies that $t \models last(Q[k,n]) = v \wedge after(v{:}13)$. Because $t$ and $u$ are consecutive states and $u \models last(Q[k,n]) = v$, this implies that $u \models last(Q[k,n]) = v \wedge after(v{:}13)$. Therefore, by Lemma 1, $Q[k,n]$ has the same value at both states $t$ and $u$. As stated previously, $t \models Q[k,n].seq[j] = v!seq[j]$. Hence, by (7),

$$u \models Q[k,n].wc = 2 \wedge Q[k,n].flag[j] \wedge Q[k,n].seq[j] = v!seq[j] \quad . \tag{10}$$

By (9), this implies that $u \models Q[k,n].seq[j] = w!seq[j]$. By (5) and Lemma 1, $u \models Q[k,j].seq[j] = w!seq[j]$. Hence, $u \models Q[k,n].seq[j] = Q[k,j].seq[j]$. Thus, by (10), we get the following.

$$u \models Q[k,n].wc = 2 \wedge Q[k,n].flag[j] \wedge Q[k,n].seq[j] = Q[k,j].seq[j]$$

By the definition of $WA$, this implies that $u \models \neg WA(Q,k,j)$, which is our proof obligation.

Case 2: $u \models last(Q[k,n]) \neq v$. In this case, because $t$ and $u$ are consecutive states and $t \models last(Q[k,n]) = v$, state $u$ is reached from $t$ via the occurrence of the event $p{:}5$, where $p = succ(succ(v))$. Let $p'$ be the $k$-Write operation such that $p' = succ(v)$ and $p = succ(p')$. Note that $p'$ is a $(k,n')$-Write operation, where $n' = n \oplus M$. Because $u$ is reached from $t$ via the occurrence of $p{:}5$, $last(Q[k,n']) = p'$ at state $u$.

By (6), (8), (9), and Lemma 4, there exists a Write operation $v'$ such that $v' = pred(v)$, $w{:}5 \prec v'{:}1$, and $v'!seq[j] = w!seq[j]$. Therefore, by (9),

$$(w{:}5 \prec v'{:}1) \wedge (w!seq[j] = v'!seq[j] = v!seq[j]) \quad . \tag{11}$$

Because $v'$, $v$, $p'$, and $p$ are successive operations of the same Writer, $v'{:}1 \prec p'{:}1 \prec p'{:}13 \prec p{:}5$. Therefore, by (11), the following precedence assertion holds.

$$w{:}5 \prec p'{:}1 \prec p'{:}13 \prec p{:}5 \tag{12}$$

By (12), $p' \neq w$. Thus, because $u \models last(Q[k,n']) = p'$, by (5), $n' \neq j$.

We now establish that $p'!flag[j] \wedge p'!seq[j] = w!seq[j]$. Because $last(Q[k,n']) = p'$ at state $u$ (i.e., the state following $p{:}5$), (12) implies that

$$u \models Q[k,n'].wc = 2 \quad . \tag{13}$$

By (5), $last(Q[k,j]) = w$ at state $u$; hence, by (12), $last(Q[k,j]) = w$ at the state prior to the event $p'{:}1$. Therefore, by Lemma 1, $Q[k,j].seq[j] = w!seq[j]$ at the state prior to $p'{:}1$. Therefore, by the program for the Writer, $p'!x[k,j].seq[j] = w!seq[j]$. Because $p'$ is a $(k,n')$-Write operation and $n' \neq j$, we have $p'!seq[j] = p'!x[k,j].seq[j]$. Therefore, $p'!seq[j] = w!seq[j]$. Hence, by (11), $p'!seq[j] = v!seq[j] = v'!seq[j]$. Thus, $p'!flag[j]$ holds. This establishes the following assertion.

$$p'!flag[j] \wedge p'!seq[j] = w!seq[j] \tag{14}$$

We now establish our proof obligation, i.e., $u \models \neg WA(Q,k,j)$. Because $u \models last(Q[k,j]) = w \wedge last(Q[k,n']) = p'$, by Lemma 1, $u \models Q[k,n'].flag[j] = p'!flag[j] \wedge Q[k,n'].seq[j] = p'!seq[j] \wedge Q[k,j].seq[j] = w!seq[j]$. By (13) and (14), this implies that the following assertion holds:

$$u \models Q[k,n'].wc = 2 \wedge Q[k,n'].flag[j] \wedge Q[k,n'].seq[j] = Q[k,j].seq[j]$$

where (as shown earlier) $n' \neq j$. Hence, by the definition of $WA$, $u \models \neg WA(Q,k,j)$. □

According to the following lemma, if a completed Write operation is not "strongly alive" at some state, then it is forever after not "strongly alive."

**Lemma 8:** $\neg salive(w,k) \wedge after(w{:}13)$ is stable.

**Proof:** Let $t$ and $u$ be consecutive states such that $t \models \neg salive(w,k) \wedge after(w{:}13)$. Our proof obligation is to show that $u \models \neg salive(w,k)$.

Assume that $w$ is a $(k,j)$-Write operation. By the definition of *salive*, if $u \models last(Q[k,j]) \neq w$, then $u \models \neg salive(w,k)$. So, in this case, our proof obligation is satisfied. In the remainder of the proof, assume that $u \models last(Q[k,j]) = w$.

Because $t$ and $u$ are consecutive states and $t \models after(w{:}13)$ and $u \models last(Q[k,j]) = w$, the following assertion holds at both states $t$ and $u$.

$$last(Q[k,j]) = w \wedge after(w{:}13)$$

Therefore, because $salive(w,k)$ is false at $t$,

$$t \models last(Q[k,j]) = w \wedge after(w{:}13) \wedge \neg salive(w,k) \quad . \tag{15}$$

From (15) and Lemma 1, we have $t \models Q[k,j].wc = 2$. From (15) and the definition of *salive*, we also have $t \models \neg SA(Q,k,j)$. Therefore,

$$t \models Q[k,j].wc = 2 \wedge \neg SA(Q,k,j) \ .$$

Thus, by the definition of $SA$, there are two possibilities to consider: $t \models \neg WA(Q,k,j)$ or $t \models Q[k,j].match[1]$. In the former case, by (15), we have $t \models last(Q[k,j]) = w \wedge \neg WA(Q,k,j) \wedge after(w{:}13)$. Hence, by the definition of *walive*, $t \models \neg walive(w,k) \wedge after(w{:}13)$. Therefore, by Lemma 7, $u \models \neg walive(w,k)$. Hence, by the contrapositive of Lemma 2, $u \models \neg salive(w,k)$, which establishes our proof obligation.

Now, consider the latter case, i.e., $t \models Q[k,j].match[1]$. As established above, $last(Q[k,j]) = w \wedge after(w{:}13)$ holds at both states $t$ and $u$. By Lemma 1, this implies that $Q[k,j].match[1]$ has the same value at both states $t$ and $u$. Therefore, because $Q[k,j].match[1]$ holds at $t$, it also holds at $u$. Hence, by the definition of $SA$, $u \models \neg SA(Q,k,j)$. Thus, by the definition of *salive*, $u \models \neg salive(w,k)$. □

The next lemma gives the conditions under which $salive(v,k)$ may be falsified.

**Lemma 9:** Suppose that $t$ and $u$ are consecutive states such that $t \models salive(v,k)$ and $u \models \neg salive(v,k)$. Then, there exists a $k$-Write operation $w$ such that $v{:}11 \prec w{:}5$ and $u \models after(w{:}13)$.

**Proof:** Let $t$, $u$, and $v$ be as defined in the lemma. Assume that $v$ is a $(k,j)$-Write operation. Let $e$ be the event prior to state $u$, i.e., $t \xrightarrow{e} u$. We first dispose of the case in which $u \models last(Q[k,j]) \neq v$. Because $t \models salive(v,k)$, by the definition of *salive*, $t \models last(Q[k,j]) = v$. Because $t$ and $u$ are consecutive states and $last(Q[k,j]) = v$ holds at $t$ but not $u$, event $e$ equals $v'{:}5$, where $v' = succ(succ(v))$. Let $w$ be the $k$-Write operation such that $w = succ(v)$ and $v' = succ(w)$. Because $w = succ(v)$, we have $v{:}11 \prec w{:}5$. Because $v' = succ(w)$ and event $e$ equals $v'{:}5$, we have $u \models after(w{:}13)$. So, in this case, our proof obligation is satisfied.

In the remainder of the proof, we assume that $u \models last(Q[k,j]) = v$. We first show that $u \models Q[k,j].wc = 2 \wedge \neg Q[k,j].match[0] \wedge \neg Q[k,j].match[1]$. Because $t \models salive(v,k)$, by the definition of *salive*, $t \models last(Q[k,j]) = v \wedge Q[k,j].wc = 2$. By the program for the Writer, this implies that

$$t \models last(Q[k,j]) = v \wedge after(v{:}13) \ . \tag{16}$$

Hence, by Lemma 1, $t \models Q[k,j].match[0] = v!match[0] \wedge Q[k,j].match[1] = v!match[1]$. Because $t \models salive(v,k)$, by the definition of *salive* and $SA$, $t \models \neg Q[k,j].match[0] \wedge \neg Q[k,j].match[1]$. Therefore, $v!match[0]$ and $v!match[1]$ are both false.

By assumption, $u \models last(Q[k,j]) = v$. Therefore, because $t$ and $u$ are consecutive states, by (16) and the definition of *after*, $u \models last(Q[k,j]) = v \wedge after(v{:}13)$. Hence,

by Lemma 1, $u \models Q[k,j].wc = 2 \wedge Q[k,j].match[0] = v!match[0] \wedge Q[k,j].match[1] = v!match[1]$. Therefore, because $v!match[0]$ and $v!match[1]$ are both false, $u \models Q[k,j].wc = 2 \wedge \neg Q[k,j].match[0] \wedge \neg Q[k,j].match[1]$.

To recapitulate, we have $u \models last(Q[k,j]) = v \wedge Q[k,j].wc = 2 \wedge \neg Q[k,j].match[0] \wedge \neg Q[k,j].match[1]$. Because $salive(v,k)$ does not hold at $u$, this implies that there exists $n$, where $n \neq j$, such that the following expression holds.

$$u \models Q[k,n].wc = 2 \wedge Q[k,n].flag[j] \wedge Q[k,n].seq[j] = Q[k,j].seq[j] \qquad (17)$$

Let $w$ be the Write operation such that $u \models last(Q[k,n]) = w$. ($w$ exists because $Q[k,n].flag[j]$ is initially false.) Then, because $u \models Q[k,n].wc = 2$, by the program for the Writer, $u \models after(w{:}13)$. This establishes one of our proof obligations.

We now establish our other proof obligation, i.e., $v{:}11 \prec w{:}5$. Assume, to the contrary, that $w{:}5 \prec v{:}11$. By (16), $after(v{:}13)$ holds at state $t$, i.e., the state prior to $e$. This implies that $v{:}11 \prec e$. Therefore,

$$w{:}5 \prec v{:}11 \prec e \ .$$

Let $t'$ be the state prior to $v{:}11$. Because $last(Q[k,n]) = w$ at state $u$ (i.e., the state following $e$), the above precedence assertion implies that $last(Q[k,n]) = w$ at state $t'$. Therefore, by Lemma 1,

$$t' \models Q[k,n].flag[j] = w!flag[j] \wedge Q[k,n].seq[j] = w!seq[j] \ . \qquad (18)$$

Because $u \models last(Q[k,j]) = v \wedge last(Q[k,n]) = w$, by Lemma 1, $u \models Q[k,n].flag[j] = w!flag[j] \wedge Q[k,n].seq[j] = w!seq[j] \wedge Q[k,j].seq[j] = v!seq[j]$. Therefore, by (17), $w!flag[j]$ is true and $w!seq[j] = v!seq[j]$. Hence, by (18), $t' \models Q[k,n].flag[j] \wedge Q[k,n].seq[j] = v!seq[j]$.

By the program for the Writer, $t' \models Q = v!z$. Therefore, $v!z[k,n].flag[j] \wedge v!z[k,n].seq[j] = v!seq[j]$. By the program for the Writer, this implies that $v!match[1]$ is true. By (16) and Lemma 1, $t \models Q[k,j].match[1] = v!match[1]$. Hence, $Q[k,j].match[1]$ is true at state $t$. By the definition of $SA$, this implies that $t \models \neg SA(Q,k,j)$. But, this implies that $salive(v,k)$ is false at state $t$, which is a contradiction. Therefore, our assumption that $w{:}5 \prec v{:}11$ is false, i.e., $v{:}11 \prec w{:}5$. □

The following lemma shows that the value of the "preferable" tag does not decrease from state to state.

**Lemma 10:** Let $t$ and $u$ be consecutive states such that $t \models spref(v,k)$ and $u \models spref(w,k)$. Then, $(v!tag, v!i) \leq (w!tag, w!i)$.

**Proof:** Let $t$, $u$, $v$, and $w$ be as defined in the lemma. If $u \models salive(v,k)$, then because

$u \models spref(w,k)$, we have $(v!tag, v!i) \leq (w!tag, w!i)$. Thus, in this case, our proof obligation is satisfied. In the remainder of the proof, assume that $u \models \neg salive(v,k)$.

Let $e$ be the event prior to state $u$, i.e., $t \xrightarrow{e} u$. Because $t \models spref(v,k)$, by the definition of *spref*, $t \models salive(v,k)$. Therefore, by Lemma 9, there exists a $k$-Write operation $w'$ such that $v{:}11 \prec w'{:}5$ and $u \models after(w'{:}13)$.

Let $S$ be the set of $k$-Write operations defined as follows: $p \in S$ iff $p$ is a $k$-Write operation and $u \models after(p{:}13)$. Observe that $w'$ is in $S$. Let $w''$ denote the Write operation in $S$ such that for each other Write operation $p$ in $S$, $p{:}5 \prec w''{:}5$. Then, by Lemma 5,

$$u \models salive(w'', k) \quad . \tag{19}$$

Because $w'$ is in $S$, $w'{:}5 \preceq w''{:}5$. By the program for the Writer, $v{:}10 \prec v{:}11$ and $w''{:}5 \prec w''{:}13$. Because $w''$ is in $S$, $after(w''{:}13)$ holds at $u$; hence, $w''{:}13 \preceq e$. Therefore,

$$v{:}10 \prec v{:}11 \prec w'{:}5 \preceq w''{:}5 \prec w''{:}13 \preceq e \quad . \tag{20}$$

Let $t'$ be the state prior to $w''{:}6$. We now show that $t' \models walive(v,k)$. Because $t \models salive(v,k)$, by Lemma 2, $t \models walive(v,k)$. Therefore, by Lemma 7, $walive(v,k)$ is true for all states between $v{:}10$ and $e$. Thus, by (20), $walive(v,k)$ is true at state $t'$.

We now show that $w!tag > v!tag$. Assume that $v$ is a $(k,j)$-Write operation. Because $t' \models walive(v,k)$, by the definition of *walive*, $t' \models last(Q[k,j]) = v \wedge WA(Q,k,j)$. By (20) and the definition of state $t'$, we have $t' \models after(v{:}10)$. Therefore, because $t' \models last(Q[k,j]) = v \wedge after(v{:}10)$, by Lemma 1, $t' \models Q[k,j].tag = v!tag$. This establishes the following assertion.

$$t' \models Q[k,j].tag = v!tag \wedge WA(Q,k,j)$$

Because $WA(Q,k,j)$ holds at $t'$, by the program for the Writer, $t' \models w''!tag > Q[k,j].tag$. Therefore, $w''!tag > v!tag$. Because $u \models spref(w,k)$, by (19) and the definition of *spref*, $w!tag \geq w''!tag$. Hence, $w!tag > v!tag$. This implies that $(v!tag, v!i) \leq (w!tag, w!i)$, which establishes our proof obligation. □

According to the next lemma, if a completed Write operation is not "preferable" at some state, then it is forever after not "preferable."

**Lemma 11:** $\neg spref(w,k) \wedge \text{after}(w{:}13)$ is stable.

**Proof:** Let $t$ and $u$ be consecutive states such that $t \models \neg spref(w,k) \wedge after(w{:}13)$. By the definition of *after*, $u \models after(w{:}13)$. This reduces our proof obligation to showing $u \models \neg spref(w,k)$.

If $salive(w,k)$ is false at $u$, then by the definition of *spref*, $spref(w,k)$ is also false at $u$, which establishes our proof obligation. So, in the remainder of the proof, assume that $salive(w,k)$ holds at $u$. By Lemma 8, this implies that $salive(w,k)$ holds at $t$ as well.

Because $after(w{:}13)$ holds at both states $t$ and $u$, by Lemma 6 and the definition of $spref$, there exist $k$-Write operations $v$ and $v'$ such that $t \models spref(v,k)$ and $u \models spref(v',k)$. Because $t \models salive(w,k) \wedge \neg spref(w,k)$, by the definition of $spref$, $(w!tag, w!i) < (v!tag, v!i)$. Because $t$ and $u$ are consecutive states, by Lemma 10, $(v!tag, v!i) \leq (v'!tag, v'!i)$. Therefore, by transitivity, $(w!tag, w!i) < (v'!tag, v'!i)$. Hence, because $u \models spref(v',k)$, by the definition of $spref$, $u \models \neg spref(w,k)$. □

According to the next lemma, if Read operation $r$ Reads the value Written by $k$-Write operation $w$, then no other $k$-Write operation "lies between" $w$ and $r$.

**Lemma 12:** Suppose that $spref(w,k)$ holds at the state prior to $r{:}0$. Then, for each $k$-Write operation $w'$ that differs from $w$, $w'{:}0 \prec w{:}13$ or $r{:}0 \prec w'{:}13$.

**Proof:** Let $r$ and $w$ be as defined in the lemma. Assume that $w$ is a $(k,j)$-Write operation. Let $t$ be the state prior to $r{:}0$. Because $t \models spref(w,k)$, by the definition of $spref$, $t \models salive(w,k)$. By the definition of $salive$, this implies that $t \models last(Q[k,j]) = w \wedge Q[k,j].wc = 2$. Therefore, by the program for the Writer, $w{:}13 \prec r{:}0$. We establish our proof obligation by assuming, to the contrary, that there exists a $(k,n)$-Write operation $w'$ (that differs from $w$) such that

$$w{:}13 \prec w'{:}0 \prec w'{:}13 \prec r{:}0 \quad . \tag{21}$$

Because $last(Q[k,j]) = w$ at state $t$, this precedence assertion implies that $j \neq n$.

Let $S$ be the set of $k$-Write operations defined as follows: $p \in S$ iff $p$ is a $k$-Write operation and $after(p{:}13)$ holds at state $t$ (i.e., the state prior to $r{:}0$). By (21), $w'$ is in $S$. Let $v$ be the Write operation in $S$ such that for each other Write operation $p$ in $S$, $p{:}5 \prec v{:}5$. Then, by Lemma 5, $t \models salive(v,k)$.

Let $t'$ be the state prior to $v{:}6$. We now show that $walive(w,k)$ holds at state $t'$. Because $w'$ is in $S$, $w'{:}5 \preceq v{:}5$. Hence, by (21), $w{:}13 \prec v{:}5$. By the program for the Writer, $v{:}5 \prec v{:}13$. Because $v$ is in $S$, $v{:}13 \prec r{:}0$. Therefore, $w{:}13 \prec v{:}5 \prec v{:}13 \prec r{:}0$. Because $salive(w,k)$ holds at state $t$, by Lemma 8, this precedence assertion implies that $salive(w,k)$ holds at state $t'$. Hence, by Lemma 2, $walive(w,k)$ also holds at state $t'$.

We now show that $v!tag > w!tag$. Because $t' \models walive(w,k)$, by the definition of $walive$, $t' \models last(Q[k,j]) = w \wedge WA(Q,k,j)$. Because $w{:}13 \prec v{:}5$, by the definition of state $t'$, we have $t' \models after(w{:}10)$. Because $t' \models last(Q[k,j]) = w \wedge after(w{:}10)$, by Lemma 1, $t' \models Q[k,j].tag = w!tag$. This establishes the following assertion.

$$t' \models Q[k,j].tag = w!tag \wedge WA(Q,k,j)$$

Because $WA(Q,k,j)$ holds at $t'$, by the program for the Writer, $t' \models v!tag > Q[k,j].tag$. Therefore, $v!tag > w!tag$.

Because $t \models salive(v,k)$ and $v!tag > w!tag$, by the definition of $spref$, $t \models \neg spref(w,k)$. But, by the statement of the lemma, $t \models spref(w,k)$. Therefore, we have a contradiction. Hence, our assumption that there exists $w'$ such that $w{:}13 \prec w'{:}0 \prec w'{:}13 \prec r{:}0$ is false. This establishes our proof obligation. □

The following lemma is a strengthening of the Integrity condition.

**Lemma 13:** Let $r$ be a Read operation. Then, there exists a $(k,j)$-Write operation $w$ such that (i) $\phi_k(r) = \phi_k(w) = w!eid$, (ii) $r!val[k] = w!val$, (iii) $last(Q[k,j]) = w \wedge Q[k,j].eid = w!eid$ holds at the state prior to $r{:}0$, and (iv) $w{:}13 \prec r{:}0$ and $spref(w,k)$ holds at each state between $w{:}13$ and $r{:}0$.

**Proof:** Let $r$ be a Read operation and let $t$ be the state prior to the event $r{:}0$. By the definition of $\phi_k$, $\phi_k(r) = r!id[k]$. Let $j = r!max[k]$. (As explained on page 18, Lemma 6 implies that $r!max[k]$ is well-defined.) By the program for the Reader, $t \models Q[k,j].eid = r!id[k] \wedge Q[k,j].val = r!val[k]$. Hence, by transitivity,

$$t \models Q[k,j].eid = \phi_k(r) \wedge Q[k,j].val = r!val[k] \quad . \tag{22}$$

Also, because $j = r!max[k]$, by the program for the Reader,

$$t \models SA(Q,k,j) \wedge (\forall n : SA(Q,k,n) : (Q[k,n].tag, n) \leq (Q[k,j].tag, j)) \quad . \tag{23}$$

We now show that the value of $Q[k,j]$ at state $t$ differs from its initial value. By our assumption concerning the initial Writes, there exists a $k$-Write operation $p$ such that $after(p{:}13)$ holds at $t$. By Lemma 6, this implies that there exists a $(k,j')$-Write operation $p'$ such that $t \models salive(p',k)$. It can be shown that each $tag$ field in the construction is always nonnegative. Thus, $p'!tag = p'!y[k,l].tag + 1 > 0$, where $l = p'!max$. Because $t \models salive(p',k)$, we have $t \models last(Q[k,j']) = p' \wedge SA(Q,k,j')$. By the definition of $SA$, this implies that $t \models last(Q[k,j']) = p' \wedge Q[k,j'].wc = 2$. Hence, by the program for the Writer, $t \models last(Q[k,j']) = p' \wedge after(p'{:}13)$. By Lemma 1, this implies that $t \models Q[k,j'].tag = p'!tag$. Therefore, $t \models SA(Q,k,j') \wedge Q[k,j'].tag > 0$. Hence, by (23), $t \models Q[k,j].tag \geq Q[k,j'].tag > 0$. Thus, the value of $Q[k,j].tag$ at state $t$ differs from its initial value.

Let $w$ be the $k$-Write operation such that $t \models last(Q[k,j]) = w$. ($w$ exists because the value of $Q[k,j]$ at state $t$ differs from its initial value.) By (23), $t \models SA(Q,k,j)$. Therefore, by the definition of $SA$, $t \models Q[k,j].wc = 2$. From the program for the Writer, this implies that $t \models after(w{:}13)$. Therefore, by Lemma 1,

$$\begin{aligned} t \models\ & last(Q[k,j]) = w \wedge after(w{:}13) \wedge Q[k,j].eid = w!eid \wedge \\ & Q[k,j].val = w!val \wedge Q[k,j].tag = w!tag \quad . \end{aligned} \tag{24}$$

Observe that (24) establishes (iii).

We now show that $t \models spref(w,k)$. By (23) and (24), we have $t \models last(Q[k,j]) = w \wedge SA(Q,k,j)$. Therefore, by the definition of *salive*, $t \models salive(w,k)$. By the definition of *spref*, we are thus obligated to show that if $t \models salive(v,k)$, where $v$ is a $(k,n')$-Write operation, then $(v!tag, n') \leq (w!tag, j)$. So, assume that $v$ is a $(k,n')$-Write operation and that $t \models salive(v,k)$. By (24),

$$t \models Q[k,j].tag = w!tag \quad . \tag{25}$$

Because $t \models salive(v,k)$, by the definition of *salive*, $t \models last(Q[k,n']) = v \wedge SA(Q,k,n')$. By the definition of $SA$, this implies that $t \models last(Q[k,n']) = v \wedge Q[k,n'].wc = 2$. Hence, by the program for the Writer, $t \models last(Q[k,n']) = v \wedge after(v{:}13)$. Thus, by Lemma 1, $t \models Q[k,n'].tag = v!tag$. Therefore, we have $t \models SA(Q,k,n') \wedge Q[k,n'].tag = v!tag$. Consequently, by (23) and (25), $(v!tag, n') \leq (w!tag, j)$. This establishes that $t \models spref(w,k)$.

By (24), $after(w{:}13)$ holds at state $t$ (i.e., the state prior to $r{:}0$). Therefore, $w{:}13 \prec r{:}0$. Since $spref(w,k)$ holds at state $t$, by Lemma 11, $spref(w,k)$ holds at each state between $w{:}13$ and $r{:}0$. This establishes condition (iv). In particular, $spref(w,k)$ holds at the state following $w{:}13$. Therefore, by the definition of $\phi_k$, $\phi_k(w) = w!eid$. By (24), $t \models Q[k,j].eid = w!eid \wedge Q[k,j].val = w!val$. Hence, by (22), $\phi_k(r) = \phi_k(w)$ and $r!val[k] = w!val$. This establishes conditions (i) and (ii). □

We now use the preceding lemmas to establish the conditions of Uniqueness, Integrity, Proximity, Read Precedence, and Write Precedence.

**Proof of Uniqueness:** Uniqueness is satisfied since the shared auxiliary variable $P[k]$ is atomically incremented whenever a $k$-Write operation assigns its value to either private variable *bid* or *eid*. □

**Proof of Integrity:** Integrity follows from conditions (i) and (ii) of Lemma 13. □

**Proof of Proximity:** Let $r$ be a Read operation and let $w$ be a $k$-Write operation. We prove that Proximity is satisfied by proving the stronger result $r{:}0 \prec w{:}0 \Rightarrow \phi_k(r) < \phi_k(w)$ and $w{:}13 \prec r{:}0 \Rightarrow \phi_k(w) \leq \phi_k(r)$.

Let $t$ denote the state prior to the event $r{:}0$, let $u$ denote the state prior to the event $w{:}0$, and let $u'$ denote the state prior to the event $w{:}13$.

Case 1: $r{:}0 \prec w{:}0$. By the definition of $\phi_k$, either $u \models \phi_k(w) = P[k]$ or $u' \models \phi_k(w) = P[k]$. Because $r{:}0 \prec w{:}0$, state $t$ occurs before both states $u$ and $u'$. Notice that a Write operation only changes the value of $P[k]$ by atomically incrementing it; thus, the value of $P[k]$ at either state $u$ or $u'$ is at least the value of $P[k]$ at state $t$. Therefore, $t \models P[k] \leq \phi_k(w)$.

By conditions (i) and (iii) of Lemma 13, there exists $j$ such that $t \models Q[k,j].eid = \phi_k(r)$.

Because $P[k]$ is incremented atomically when a Write operation assigns its value to either of its private variables *bid* or *eid*, $t \models Q[k,j].eid < P[k]$. Therefore, by transitivity, $\phi_k(r) < \phi_k(w)$.

Case 2: $w{:}13 \prec r{:}0$. By condition (i) of Lemma 13, there exists a $k$-Write operation $v$ such that $\phi_k(r) = \phi_k(v) = v!eid$. Moreover, condition (iv) of Lemma 13 implies that $v{:}13 \prec r{:}0$ and $spref(v,k)$ holds at each state between $v{:}13$ and $r{:}0$.

We now meet our proof obligation by showing that $\phi_k(w) \leq \phi_k(v)$. If $v = w$, then the result trivially holds, so assume that $v \neq w$. Then, by Lemma 12, $w{:}0 \prec v{:}13$. If $w{:}13 \prec v{:}13$, then $w!bid < v!eid$ and $w!eid < v!eid$; thus, by the definition of $\phi_k$, $\phi_k(w) < \phi_k(v)$. Now consider the other case, i.e., $w{:}0 \prec v{:}13 \prec w{:}13 \prec r{:}0$. In this case, $v!i \neq w!i$. Thus, because $spref(v,k)$ holds for all states between $v{:}13$ and $r{:}0$, by the definition of *spref*, $spref(w,k)$ is false at the state following $w{:}13$. Hence, $\phi_k(w) = w!bid$. Because $w{:}0 \prec v{:}13$, $w!bid < v!eid$. Therefore, $\phi_k(w) < \phi_k(v)$. □

**Proof of Read Precedence:** Let $r$ and $s$ be two Read operations. We prove that Read Precedence holds by proving $r{:}0 \prec s{:}0 \Rightarrow (\forall k :: \phi_k(r) \leq \phi_k(s))$.

Assume that $r{:}0 \prec s{:}0$. By conditions (i) and (iv) of Lemma 13, there exists a $k$-Write operation $w$ such that $\phi_k(w) = \phi_k(r)$ and $w{:}13 \prec r{:}0$. Hence, by transitivity, $w{:}13 \prec s{:}0$. By the proof of Proximity, this implies that $\phi_k(w) \leq \phi_k(s)$. Therefore, $\phi_k(r) \leq \phi_k(s)$. □

**Proof of Write Precedence:** Let $r$ be a Read operation, let $v$ be a $j$-Write operation, and let $w$ be a $k$-Write operation. Assume that $v$ precedes $w$ and $\phi_k(w) \leq \phi_k(r)$. Our proof obligation is to show that $\phi_j(v) \leq \phi_j(r)$.

In the proof of Proximity, we showed that $r{:}0 \prec w{:}0 \Rightarrow \phi_k(r) < \phi_k(w)$. By the contrapositive of this expression and by our assumption that $\phi_k(w) \leq \phi_k(r)$, we conclude that $w{:}0 \prec r{:}0$. Because $v$ precedes $w$, $v{:}13 \prec w{:}0$. Thus, by transitivity, $v{:}13 \prec r{:}0$. By the proof of Proximity, this implies that $\phi_j(v) \leq \phi_j(r)$. □

# 4 Concluding Remarks

Our results show that we can allow an atomic operation of a concurrent program to either write a single shared variable or read several shared variables (but not both) and the resulting program can be implemented from atomic registers. By contrast, if we allow an atomic operation of a program to either write several shared variables, or to both read and write shared variables, then, in general, such a program cannot be implemented from atomic registers. This result has been proved both by Herlihy [6] and by Anderson and Gouda [3].

The problem of constructing a $K/L/M/N$ composite register has been solved independently by Afek et al. [1, 10]. Our $K/L/M/N$ construction differs from that of Afek et al. because it does not use any multiple-writer atomic registers. In particular, the $K/L/M/N$

construction presented in this paper is based upon the $K/L/1/N$ construction presented in [2], and this latter construction uses only single-writer atomic registers. It follows, then, that our construction can be used to implement a multiple-writer atomic register (the case in which there is only one component) from single-writer atomic registers.

**Acknowledgements:** I would like to thank Anish Arora, Ken Calvert, Mohamed Gouda, and Jacob Kornerup for their comments on an earlier draft of this paper.

# Appendix: Bounding the Tags

In this appendix, we show that is possible to bound the size of the *tag* fields. As seen in Figure 5, a Write operation compares the *tag* fields of two different elements of $Q$ only if both elements are weakly alive. Also, as seen in Figure 4, a Read operation compares the *tag* fields of two different elements of $Q$ only if both elements are strongly alive. From the definition of $SA$, if an element of $Q$ is strongly alive, then it is also weakly alive. Therefore, it suffices to prove that the *tag* fields of the weakly alive elements of a particular component are within some bounded range.

The basic insight is really rather simple. In particular, consider a $(k, j)$-Write operation $w$. Note that the maximum tag for the weakly alive elements of component $k$ can increase by a "large" amount between $w$'s second read and second write of $Q$ (i.e., while $w$ is computing its tag) only if a "large" number of $k$-Write operations occur in this interval. But, in this case, the primary sequence number for $w$ will be read by at least three successive operations of some Writer, and $Q[k, j]$ will not be weakly alive after $w$'s second write to $Q$. So, if $Q[k, j]$ *is* weakly alive after $w$'s second write to $Q$, then the value of its *tag* field will differ from that of some other weakly alive element by only a "small" amount.

In the rest of this appendix, we show that the tags for the weakly alive elements of a given component are within a range of size $4M - 1$. More specifically, we prove that the following expression holds.

$$(WA(Q, k, i) \ \wedge \ WA(Q, k, j)) \ \Rightarrow \ (|Q[k, i].tag - Q[k, j].tag| \leq 4M - 2)$$

Therefore, if the smallest *tag* field among the weakly alive elements for some component is $b$, then the *tag* fields for these elements lie within the range $b, \ldots, b + 4M - 2$. This implies that we can restrict the size of each *tag* field to range over $0..8M - 4$.

We first prove a lemma that is similar to Lemma 9. This lemma gives the conditions under which $walive(v, k)$ may be falsified.

**Lemma 14:** Suppose that $t$ and $u$ are consecutive states such that $t \models walive(v, k)$ and $u \models \neg walive(v, k)$. Then, there exists a $k$-Write operation $w$ such that $v{:}6 \prec w{:}5$ and $u \models after(w{:}13)$.

**Proof:** Let $t$, $u$, and $v$ be as defined in the lemma. Assume the $v$ is a $(k,j)$-Write operation. Let $e$ be the event prior to state $u$, i.e., $t \xrightarrow{e} u$. We first dispose of the case in which $u \models last(Q[k,j]) \neq v$. Because $t \models walive(v,k)$, by the definition of *walive*, $t \models last(Q[k,j]) = v$. Because $t$ and $u$ are consecutive states and $last(Q[k,j]) = v$ holds at $t$ but not $u$, event $e$ equals $v':5$, where $v' = succ(succ(v))$. Let $w$ be the $k$-Write operation such that $w = succ(v)$ and $v' = succ(w)$. Because $w = succ(v)$, we have $v:6 \prec w:5$. Because $v' = succ(w)$ and $e$ equals $v':5$, we have $u \models after(w:13)$. This satisfies our proof obligation.

In the remainder of the proof, assume that $u \models last(Q[k,j]) = v$. We first show that $u \models Q[k,j].wc \geq 1 \wedge \neg Q[k,j].match[0]$. Because $t \models walive(v,k)$, by the definition of *walive*, $t \models last(Q[k,j]) = v \wedge Q[k,j].wc \geq 1$. By the program for the Writer, this implies that

$$t \models last(Q[k,j]) = v \wedge after(v:10) \quad . \tag{26}$$

Hence, by Lemma 1, $t \models Q[k,j].match[0] = v!match[0]$. Because $t \models walive(v,k)$, by the definition of *walive* and *WA*, $t \models \neg Q[k,j].match[0]$. Therefore, $v!match[0]$ is false.

By assumption, $u \models last(Q[k,j]) = v$. Therefore, because $t$ and $u$ are consecutive states, by (26) and the definition of *after*, $u \models last(Q[k,j]) = v \wedge after(v:10)$. Hence, by Lemma 1, $u \models Q[k,j].wc \geq 1 \wedge Q[k,j].match[0] = v!match[0]$. Therefore, because $v!match[0]$ is false, $u \models Q[k,j].wc \geq 1 \wedge \neg Q[k,j].match[0]$.

To recapitulate, we have $u \models last(Q[k,j]) = v \wedge Q[k,j].wc \geq 1 \wedge \neg Q[k,j].match[0]$. Because $walive(v,k)$ does not hold at $u$, this implies that there exists $n$, where $n \neq j$, such that the following expression holds.

$$u \models Q[k,n].wc = 2 \wedge Q[k,n].flag[j] \wedge Q[k,n].seq[j] = Q[k,j].seq[j] \tag{27}$$

Let $w$ be the Write operation such that $u \models last(Q[k,n]) = w$. ($w$ exists because $Q[k,n].flag[j]$ is initially false.) Then, because $u \models Q[k,n].wc = 2$, by the program for the Writer, $u \models after(w:13)$. This establishes one of our proof obligations.

We now establish our other proof obligation, i.e., $v:6 \prec w:5$. Assume, to the contrary, that $w:5 \prec v:6$. By (26), $v:10 \prec e$. This implies that $v:6 \prec e$. Therefore,

$$w:5 \prec v:6 \prec e \quad .$$

Let $t'$ be the state prior to $v:6$. Because $last(Q[k,n]) = w$ at state $u$ (i.e., the state following $e$), the above precedence assertion implies that $last(Q[k,n]) = w$ at state $t'$. Therefore, by Lemma 1,

$$t' \models Q[k,n].flag[j] = w!flag[j] \wedge Q[k,n].seq[j] = w!seq[j] \quad . \tag{28}$$

Because $u \models last(Q[k,j]) = v \wedge last(Q[k,n]) = w$, by Lemma 1, $u \models Q[k,n].flag[j] = w!flag[j] \wedge Q[k,n].seq[j] = w!seq[j] \wedge Q[k,j].seq[j] = v!seq[j]$. Therefore, by (27), $w!flag[j]$ is true and $w!seq[j] = v!seq[j]$. Hence, by (28), $t' \models Q[k,n].flag[j] \wedge Q[k,n].seq[j] = v!seq[j]$.

By the program for the Writer, $t' \models Q = v!y$. Therefore, $v!y[k,n].flag[j] \wedge v!y[k,n].seq[j] = v!seq[j]$. By the program for the Writer, this implies that $v!match[0]$ is true. By (26) and Lemma 1, $t \models Q[k,j].match[0] = v!match[0]$. Hence, $Q[k,j].match[0]$ is true at state $t$. However, by the definition of *walive* and *WA*, $t \models walive(v,k)$ implies that $t \models \neg Q[k,j].match[0]$. Hence, we have a contradiction. Therefore, our assumption that $w{:}5 \prec v{:}6$ is false, i.e., $v{:}6 \prec w{:}5$. □

**Definition:** We define the function $maxtag(k)$ as follows:

$$maxtag(k) \equiv \text{MAX}_{0 \le j < 2M}\{Q[k,j].tag \mid WA(Q,k,j)\}$$ □

As explained on page 18, Lemma 6 implies that at every state there exists some $j$ such that $WA(Q,k,j)$ holds. This implies that the function $maxtag(k)$ is well-defined at every state. The following three lemmas give us a means for determining the value of $maxtag(k)$ at a particular state.

**Lemma 15:** Let $t$ be the state prior to $v{:}6$. Then, $v!tag = maxtag(k) + 1$ at state $t$.

**Proof:** Let $t$ and $v$ be as defined in the lemma and let $j = v!max$. (As explained on page 18, Lemma 6 implies that $v!max$ is well-defined.) Then, by the program for the Writer,

$$t \models WA(Q,k,j) \wedge (\forall n : WA(Q,k,n) : (Q[k,n].tag, n) \le (Q[k,j].tag, j)) \ .$$

Then, by the definition of $maxtag(k)$, $t \models maxtag(k) = Q[k,j].tag$. By the program for the Writer, $t \models v!tag = Q[k,j].tag + 1$. Therefore, $t \models v!tag = maxtag(k) + 1$. □

**Lemma 16:** $walive(v,k) \Rightarrow maxtag(k) \ge v!tag$.

**Proof:** Let $v$ be a $(k,j)$-Write operation, and assume that $walive(v,k)$ holds at some state $t$. By the definition of *walive*, $t \models last(Q[k,j]) = v \wedge WA(Q,k,j)$. By the definition of *WA*, this implies that $t \models Q[k,j].wc \ge 1$. Therefore, by the program for the Writer, $t \models after(v{:}10)$. Hence, by Lemma 1, $t \models Q[k,j].tag = v!tag$. By the definition of *maxtag*, this implies that $t \models maxtag(k) \ge v!tag$. □

**Lemma 17:** $wpref(v,k) \Rightarrow maxtag(k) = v!tag$.

**Proof:** Let $v$ be a $(k,j)$-Write operation, and assume that $wpref(v,k)$ holds at some state $t$. Then, by the definition of *wpref*,

$$t \models walive(v,k) \wedge (\forall w : walive(w,k) : (w!tag, w!i) \le (v!tag, j)) \ .$$

By the definition of *walive*, this implies that $t \models WA(Q,k,j) \wedge last(Q[k,j]) = v$. Therefore, by the definition of *WA*, $t \models Q[k,j].wc \ge 1$. From the program for the Writer,

this implies that $t \models after(v{:}10)$. Hence, by Lemma 1, $t \models Q[k,j].tag = v!tag$. This establishes the following assertion.

$$t \models WA(Q,k,j) \wedge Q[k,j].tag = v!tag \wedge (\forall w : walive(w,k) : w!tag \leq Q[k,j].tag) \quad (29)$$

Suppose that $t \models WA(Q,k,n)$ holds for some $n$. To prove that $t \models maxtag(k) = v!tag$, by (29) and the definition of $maxtag$, it suffices to prove that $t \models Q[k,n].tag \leq Q[k,j].tag$.

If there exists some $w$ such that $t \models last(Q[k,n]) = w$, then we have $t \models last(Q[k,n]) = w \wedge WA(Q,k,n)$. By the definition of $WA$ and $walive$, this implies that $t \models last(Q[k,n]) = w \wedge Q[k,n].wc \geq 1 \wedge walive(w,k)$. By the program for the Writer, this implies that $t \models last(Q[k,n]) = w \wedge after(w{:}10) \wedge walive(w,k)$. Hence, by Lemma 1, $t \models Q[k,n].tag = w!tag \wedge walive(w,k)$. Therefore, by (29), $t \models Q[k,n].tag \leq Q[k,j].tag$.

If no such $w$ exists, then the value of $Q[k,n].tag$ at state $t$ equals its initial value. Therefore, by the definition of the initial state, $t \models Q[k,n].tag = 0$. Because each $tag$ field is always nonnegative, this implies that $t \models Q[k,n].tag \leq Q[k,j].tag$. □

Lemmas 18 and 19 together give us means for determining how much the value of $maxtag(k)$ can increase over an interval of states.

**Lemma 18:** Let $t$ and $u$ be consecutive states and let $e$ be the event prior to $u$. Let $b$ be the value of $maxtag(k)$ at state $t$. Then, $maxtag(k)$ equals either $b$ or $b+1$ at state $u$, and, in the latter case, event $e$ is of the form $w{:}10$ for some $k$-Write operation $w$.

**Proof:** At suffices to prove that the lemma holds for two consecutive states $t$ and $u$, given the assumption that the lemma holds for the prefix of the given history ending with state $t$. Let $e$ be the event prior to state $u$, and let $b$ be the value of $maxtag(k)$ at state $t$. Our proof obligation is to show that $maxtag(k)$ equals either $b$ or $b+1$ at state $u$, and, in the latter case, event $e$ is of the form $w{:}10$ for some $k$-Write operation $w$.

Let $p$ be the initial $k$-Write operation. We consider two cases, depending on whether $t \models after(p{:}13)$.

Case 1: $t \models \neg after(p{:}13)$. We establish our proof obligation by showing that $maxtag(k) = 0$ at each state prior to $p{:}10$, and that $maxtag(k) = 1$ at each state in the interval between $p{:}10$ and $p{:}13$ and at the state following $p{:}13$.

Assume that $p$ is a $(k,j)$-Write operation. By the definition of the initial state and our assumption concerning the initial Writes, $(\forall j' :: Q[k,j'].tag = 0)$ holds at each state prior to $p{:}10$. Thus, $maxtag(k) = 0$ at each state in this interval.

By the definition of the initial state and our assumption concerning the initial Writes, the following assertion holds at each state between $p{:}10$ and $p{:}13$ and at the state following $p{:}13$.

$$Q[k,j].tag = 1 \wedge Q[k,j].wc \geq 1 \wedge \neg Q[k,j].match[0] \wedge$$

$$(\forall j' : j' \neq j : Q[k, j'].tag = 0 \ \wedge \ \neg Q[k, j'].flag[j])$$

By the definition of $WA$, this implies that the following assertion holds.

$$WA(Q, k, j) \ \wedge \ Q[k, j].tag = 1 \ \wedge \ (\forall j' : j' \neq j : Q[k, j'].tag = 0)$$

Therefore, by the definition of $maxtag(k)$, $maxtag(k) = 1$ at each state between $p{:}10$ and $p{:}13$ and at the state following $p{:}13$.

Case 2: $t \models after(p{:}13)$. In this case, by Lemma 6, there exists a $k$-Write operation $q$ such that $t \models salive(q, k)$. Hence, by Lemma 2, $t \models walive(q, k)$. Thus, by the definition of $wpref$, there exists a $k$-Write operation $v$ such that $t \models wpref(v, k)$. Similar reasoning shows that there exists a $k$-Write operation $w$ such that $u \models wpref(w, k)$. By Lemma 17,

$$(t \models maxtag(k) = v!tag) \ \wedge \ (u \models maxtag(k) = w!tag) \ . \tag{30}$$

We establish our proof obligation by showing that $w!tag$ equals either $v!tag$ or $v!tag + 1$, and in the latter case $e = w{:}10$.

Assume that $w$ is a $(k, n)$-Write operation. Because $u \models wpref(w, k)$, by the definition of $wpref$, $u \models walive(w, k)$. Thus, by the definition of $walive$ and $WA$, $u \models last(Q[k, n]) = w \ \wedge \ Q[k, n].wc \geq 1$. By the program for the Writer, this implies that $u \models after(w{:}10)$. This establishes the following assertion.

$$u \models walive(w, k) \ \wedge \ after(w{:}10) \ . \tag{31}$$

We now show that if $w!tag = v!tag + 1$, then $e = w{:}10$. In this case, because $wpref(v, k)$ holds at $t$, $walive(w, k)$ does not hold at $t$. By (31), $walive(w, k) \ \wedge \ after(w{:}10)$ holds at $u$. Because $t$ and $u$ are consecutive states such that $t \models \neg walive(w, k)$ and $u \models walive(w, k) \ \wedge \ after(w{:}10)$, by Lemma 7, $t \models \neg after(w{:}10)$. Therefore, because $after(w{:}10)$ is false at $t$ but true at $u$, $u$ is reached from $t$ via the occurrence of the event $w{:}10$.

We now consider our other proof obligation, i.e., $v!tag \leq w!tag \leq v!tag + 1$. Let $u'$ be the state prior to the event $w{:}6$. By Lemma 15, $u' \models w!tag = maxtag(k) + 1$. Because $t$ and $u$ are consecutive states, from (31) we conclude that $u'$ occurs before $t$. Therefore, because the lemma holds for the prefix of the given history ending with state $t$, the value of $maxtag(k)$ at state $t$ is at least its value at state $u'$. Therefore, $t \models w!tag \leq maxtag(k) + 1$. Consequently, by (30), $w!tag \leq v!tag + 1$.

Our remaining proof obligation is to show that $v!tag \leq w!tag$. We first dispose of the case in which $u \models walive(v, k)$. In this case, because $u \models wpref(w, k)$, by the definition of $wpref$, $w!tag \geq v!tag$.

In the remainder of the proof, we consider the case in which $u \models \neg walive(v, k)$. Because $t \models wpref(v, k)$, we have $t \models walive(v, k)$. Therefore, by Lemma 14, there exists a $k$-Write operation $w'$ such that $v{:}6 \prec w'{:}5$ and $u \models after(w'{:}13)$. Let $S$ denote

the set of $k$-Write operations defined as follows: $q \in S$ iff $q$ is a $k$-Write operation and $u \models \mathit{after}(q{:}13)$. Observe that $w'$ is in $S$. Let $w''$ denote the Write operation in $S$ such that for each other Write operation $q$ in $S$, $q{:}5 \prec w''{:}5$. Then, $w'{:}5 \preceq w''{:}5$. Furthermore, because $\mathit{after}(w''{:}13)$ holds at $u$, i.e., the state following event $e$, $w''{:}13 \preceq e$. Thus,

$$v{:}6 \prec w'{:}5 \preceq w''{:}5 \prec w''{:}13 \preceq e \quad . \tag{32}$$

By Lemma 5, $u \models \mathit{salive}(w'', k)$. By Lemma 2, this implies that $u \models \mathit{walive}(w'', k)$. Therefore, by Lemma 16, $u \models \mathit{maxtag}(k) \geq w''!\mathit{tag}$. By (30), this implies that

$$w''!\mathit{tag} \leq w!\mathit{tag} \quad . \tag{33}$$

Let $t'$ be the state prior to $v{:}6$ and let $t''$ be the state prior to $w''{:}6$. Then, by Lemma 15,

$$(t' \models v!\mathit{tag} = \mathit{maxtag}(k) + 1) \ \wedge \ (t'' \models w''!\mathit{tag} = \mathit{maxtag}(k) + 1) \quad . \tag{34}$$

Notice that (32) implies that $t''$ occurs between $t'$ and $t$. Therefore, because the lemma holds for the prefix of the given history ending with state $t$, the value of $\mathit{maxtag}(k)$ at state $t''$ is at least its value at state $t'$. Hence, by (34), $v!\mathit{tag} \leq w''!\mathit{tag}$. Consequently, by (33), $v!\mathit{tag} \leq w!\mathit{tag}$. □

**Lemma 19:** Suppose that state $t$ occurs before state $u$ and that $Q[k, j].\mathit{seq}[j]$ has the same value at each state in the closed interval $[t, u]$. Let $b$ be the value of $\mathit{maxtag}(k)$ at state $u$. If $u \models \mathit{WA}(Q, k, j)$, then $t \models \mathit{maxtag}(k) \geq b - 4M + 2$.

**Proof:** Let $t$, $u$, and $b$ be as defined in the statement of the lemma, and suppose that $Q[k, j].\mathit{seq}[j]$ has the same value at every state in the closed interval $[t, u]$. Assume that $u \models \mathit{WA}(Q, k, j)$.

Let $D$ denote the number of events between $t$ and $u$ of the form $v{:}10$, where $v$ is a $k$-Write operation. Then, by Lemma 18, $t \models \mathit{maxtag}(k) \geq b - D$. Therefore, to establish our proof obligation, it suffices to show that $D \leq 4M - 2$.

Let $j' = j \oplus M$. By the program for the Writer, if $v$ is a $(k, j)$-Write operation, then the value of $Q[k, j].\mathit{seq}[j]$ at the state prior to $v{:}5$ differs from its value at the state following $v{:}5$. Therefore, because $Q[k, j].\mathit{seq}[j]$ has the same value at every state in $[t, u]$, there are no events between $t$ and $u$ of the form $v{:}5$, where $v$ is a $(k, j)$-Write operation. Because successive operations of the same Writer write to different elements of $Q$, this implies that between $t$ and $u$ there is at most one event $v{:}10$, where $v$ is a $(k, j)$-Write operation, and at most one such event, where $v$ is a $(k, j')$-Write operation.

Let $n \neq j \ \wedge \ n \neq j'$, and let $n' = n \oplus M$. In the remainder of the proof, we use $v$ to denote an arbitrary $(k, n)$- or $(k, n')$-Write operation. We show that there are at most four events of the form $v{:}10$ between $t$ and $u$. Assume, to the contrary, that there are at least five such events between $t$ and $u$. Then, there exists an event of the form $v{:}4$ between $t$

and $u$. Let $v_4{:}4$ be the last such event between $t$ and $u$. Let $v_3 = pred(v_4)$, $v_2 = pred(v_3)$, $v_1 = pred(v_2)$, and $v_0 = pred(v_1)$. Let $e$ be the event following state $t$, and let $f$ be the event prior to state $u$. Then, because there are at least five events of the form $v{:}10$ between $t$ and $u$, the following precedence assertion holds.

$$e \prec v_1{:}0 \prec v_1{:}13 \prec v_2{:}0 \prec v_2{:}13 \prec v_3{:}0 \prec v_3{:}13 \prec f \tag{35}$$

Without loss of generality, assume that $v_3$ is a $(k,n)$-Write operation. Because $v_4{:}4$ is the last event between $t$ and $u$ of the form $v{:}4$ (and because successive operations of the same Writer write to different elements of $Q$), (35) implies that $v_3{:}13$ is the last event to write to $Q[k,n]$ before state $u$.

Assume that $Q[k,j].seq[j] = q$ at state $u$. Then, by assumption, $Q[k,j].seq[j] = q$ at every state in the interval $[t,u]$. By (35) and the program for the Writer, this implies that

$$v_1!seq[j] = v_2!seq[j] = v_3!seq[j] = q \quad .$$

Therefore, $v_3!flag[j]$ holds. Hence, because $v_3{:}13$ is the last event to write to $Q[k,n]$ before state $u$, $u \models Q[k,n].wc = 2 \wedge Q[k,n].flag[j] \wedge Q[k,n].seq[j] = q$. Therefore,

$$u \models Q[k,n].wc = 2 \wedge Q[k,n].flag[j] \wedge Q[k,n].seq[j] = Q[k,j].seq[j] \quad .$$

Hence, by the definition of $WA$, $u \models \neg WA(Q,k,j)$, which is a contradiction.

So, to summarize, there is at most one event between $t$ and $u$ of the form $p{:}10$, if $p$ is a $(k,j)$-Write operation; at most one such event, if $p$ is a $(k,j')$-Write operation; and at most four such events, if $p$ is either a $(k,n)$- or $(k,n \oplus M)$-Write operation, $0 \leq n < M$ and $n \neq j$ **modulo** $M$. Therefore, there are at most $2 + 4(M-1)$ such events total between $t$ and $u$. Hence, $D \leq 4M - 2$. This establishes our proof obligation. □

**Proof of Boundedness:** We prove that the following expression holds.

$$(WA(Q,k,i) \wedge WA(Q,k,j)) \Rightarrow (|Q[k,i].tag - Q[k,j].tag| \leq 4M-2)$$

Let $u$ be a state, and suppose that $u \models WA(Q,k,i) \wedge WA(Q,k,j)$, where $i \neq j$. Our proof obligation is to show that $u \models |Q[k,i].tag - Q[k,j].tag| \leq 4M-2$.

If $u \models Q[k,i].tag = 0 \wedge Q[k,j].tag = 0$, then our proof obligation is satisfied. So, without loss of generality, assume that $u \models Q[k,j].tag \neq 0$. We have two cases to consider, depending on whether $u \models Q[k,i].tag = 0$.

Case 1: $u \models Q[k,i].tag = 0$. In this case, it suffices to prove that $u \models Q[k,j].tag \leq 4M-2$.

Let $t$ be the initial state. Note that the value of $Q[k,j].tag$ at state $u$ differs from its initial value. Thus, $u$ is not the initial state, i.e., $t$ occurs before $u$.

We now show that $Q[k,i].seq[i]$ has the same value at every state in the closed interval $[t,u]$. If no $(k,i)$-Write operation exists in the given history, then clearly $Q[k,i].seq[i]$ has

the same value at every state in $[t, u]$. Otherwise, it suffices to prove that the event $v{:}5$ does not occur between $t$ and $u$, where $v$ is the initial $(k, i)$-Write operation. By the program for the Writer, each Write operation assigns a nonzero value to its *tag* field. Because $u \models Q[k, i].tag = 0$, this implies that the event $v{:}10$ occurs after $u$. By assumption, $u \models WA(Q, k, i)$. By the definition of $WA$, this implies that $u \models Q[k, i].wc \geq 1$. Therefore, because $v$ is the initial $(k, i)$-Write operation and $v{:}10$ occurs after $u$, by the program for the Writer, $v{:}5$ does not occur between $t$ and $u$.

Let $b$ be the value of $maxtag(k)$ at state $u$. Because $u \models WA(Q, k, i)$ and because $Q[k, i].seq[i]$ has the same value at every state in $[t, u]$, by Lemma 19, $t \models maxtag(k) \geq b - 4M + 2$. By the definition of the initial state $t \models maxtag(k) = 0$. Therefore, $b \leq 4M - 2$. Because $u \models WA(Q, k, j)$, we have $u \models Q[k, j].tag \leq b$. Therefore, $u \models Q[k, j].tag \leq 4M - 2$.

Case 2: $u \models Q[k, i].tag \neq 0$. In this case, the value of $Q[k, i].tag$ at state $u$ differs from its initial value. Therefore, there exists a $(k, i)$-Write operation $v$ such that $u \models last(Q[k, i]) = v$. Similarly, because $u \models Q[k, j].tag \neq 0$, there exists a $(k, j)$-Write operation $w$ such that $u \models last(Q[k, j]) = w$.

Because $u \models WA(Q, k, i)$, we have $u \models Q[k, i].wc \geq 1$. Therefore, because $u \models last(Q[k, i]) = v$, by the program for the Writer, $u \models after(v{:}10)$. Similarly, because $u \models WA(Q, k, j) \wedge last(Q[k, j]) = w$, we have $u \models after(w{:}10)$. This establishes the following assertion.

$$u \models last(Q[k, i]) = v \wedge after(v{:}10) \wedge last(Q[k, j]) = w \wedge after(w{:}10) \qquad (36)$$

From (36) and Lemma 1, $u \models Q[k, i].tag = v!tag \wedge Q[k, j].tag = w!tag$. Therefore, we can establish our proof obligation by showing that $|v!tag - w!tag| \leq 4M - 2$.

Without loss of generality, assume that $v{:}6 \prec w{:}6$. Let $e$ be the event prior to state $u$. By the program for the Writer, $w{:}6 \prec w{:}10$. By (36), $after(w{:}10)$ holds at state $u$; hence, $w{:}10 \preceq e$. Therefore, we have the following precedence assertion.

$$v{:}6 \prec w{:}6 \prec w{:}10 \preceq e \qquad (37)$$

Let $t$ be the state prior to $v{:}6$, and let $t'$ be the state prior to $w{:}6$. Because $last(Q[k, i]) = v$ at state $u$ (i.e., the state following event $e$), (37) implies that $last(Q[k, i]) = v$ at each state in the closed interval $[t, u]$. Therefore, by Lemma 1, $Q[k, i].seq[i]$ has the same value at every state in $[t, u]$. Let $b$ be the value of $maxtag(k)$ at state $t$, and let $c$ be the value of $maxtag(k)$ at state $u$. Then, by Lemma 19, $b \geq c - 4M + 2$. Let $b'$ be the value of $maxtag(k)$ at state $t'$. By (37), $t'$ occurs between $t$ and $u$. Therefore, by Lemma 18, $b \leq b' \leq c$. This implies that $b \leq b' \leq b + 4M - 2$. By Lemma 15, $v!tag = b + 1$, and $w!tag = b' + 1$. Therefore, $v!tag \leq w!tag \leq v!tag + 4M - 2$. □

# References


[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, Atomic snapshots, *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing*, 1990, to appear.

[2] J. Anderson, Composite registers, *Technical Report TR.89.25*, Department of Computer Sciences, University of Texas at Austin, 1989.

[3] J. Anderson and M. Gouda, The virtue of patience: concurrent programming with and without waiting, unpublished manuscript.

[4] B. Bloom, Constructing two-writer atomic registers, *IEEE Transactions on Computers*, vol. 37, no. 12, December 1988, pp. 1506-1514. Also appeared in *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 249-259.

[5] J. Burns and G. Peterson, Constructing multi-reader atomic values from non-atomic values, *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 222-231.

[6] M. Herlihy, Wait-free implementation of concurrent objects, *Proceedings of the Seventh Annual Symposium on Principles of Distributed Computing*, 1988.

[7] M. Herlihy and J. Wing, Axioms for concurrent objects, *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*, 1987.

[8] A. Israeli and M. Li, Bounded time-stamps, *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, pp. 371-382, 1987.

[9] L. Lamport, On interprocess communication, parts I and II, *Distributed Computing*, vol. 1, pp. 77-101, 1986.

[10] M. Merritt, private communication, 1990.

[11] R. Newman-Wolfe, A protocol for wait-free, atomic, multi-reader shared variables, *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 232-248.

[12] G. Peterson, Concurrent reading while writing, *ACM Transactions on Programming Languages and Systems*, vol. 5, pp. 46-55, 1983.

[13] G. Peterson and J. Burns, Concurrent reading while writing II: the multi-writer case, *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, 1987.

[14] A. Singh, J. Anderson, and M. Gouda, The elusive atomic register, revisited, *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 206-221.

[15] P. Vitanyi and B. Awerbuch, Atomic shared register access by asynchronous hardware, *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, 1986.