How to Share Concurrent Wait-Free Variables

MING LI

University of Waterloo, Waterloo, Ontario, Canada

JOHN TROMP AND PAUL M. B. VITÁNYI

Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, and Universiteit van Amsterdam, Amsterdam, The Netherlands

Abstract. Sharing data between multiple asynchronous users—each of which can atomically read and write the data—is a feature that may help to increase the amount of parallelism in distributed systems. An algorithm implementing this feature is presented. The main construction of an n-user atomic variable directly from single-writer, single-reader atomic variables uses $O(n)$ control bits and $O(n)$ accesses per Read/Write running in $O(1)$ parallel time.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles; B.4.3 [**Input/Output and Data Communications**]: Interconnections (Subsystems); D.4.1 [**Operating Systems**]: Process Management; D.4.4 [**Operating Systems**]: Communications Management

General Terms: Management

Additional Key Words and Phrases: Atomicity, concurrent reading and writing, multi-writer, shared variable (register)

1. *Introduction*

Lamport [1986] has shown how an atomic variable—one whose accesses appear to be indivisible—shared between one writer and one reader, acting asynchronously and without waiting, can be constructed from lower level hardware rather

---

M. Li was supported in part by the National Science Foundation under Grant DCR 86-06366 at Ohio State University, by Office of Naval Research Grant N00014-85-K-0445 and Army Research Office Grant DAAL03-86-K-0171 at Harvard University, and by NSERC Grant OGP-0036747 at York University.

P. Vitányi was supported in part by the European Union through NeuroCOLT ESPIRIT Working Group No. 8556, and by NWO through NFI Project ALADDIN under Contract number NF 62-376.

J. Tromp was supported in part by NWO through NFI Project ALADDIN under Contract number NF 62-376.

Authors' addresses: M. Li, Computer Science Department, University of Waterloo, Waterloo, Ont., N2L 3G1 Canada; e-mail: mli@math.uwaterloo.ca; J. Tromp and P. M. B. Vitányi, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, e-mail: tromp@cwi.nl and paulv@cwi.nl.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery (ACM), Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 1996 ACM 0004-5411/96/0700-0723 $03.50

than just assuming its existence. There arises the question of the construction of multi-user atomic variables of this type (see Vitányi and Awerbuch [1986], on which the current paper is partially based). In this paper, we will supply a uniform solution to such problems, given Lamport's construction, and derive the implementations by transformations from the specification.

1.1. INFORMAL PROBLEM STATEMENT AND MAIN RESULT. Usually, with asynchronous readers and writers, atomicity of operations is simply assumed or enforced by synchronization primitives like semaphores. However, active serialization of asynchronous concurrent actions always implies waiting by one action for another. In contrast, our aim is to realize the maximum amount of parallelism inherent in concurrent systems by avoiding waiting altogether in our algorithms. In such a setting, serializability is *not* actively enforced, rather it follows from the way the executions of the algorithm by the various processes interact. Any one of the references, say Lamport [1986] or Vitányi and Awerbuch [1986] describes the problem area in some detail. Peterson [1983] seems to be the first to precisely identify the notion of wait-free concurrent read/write variables.

The point of departure is the solution of the following problem. (We keep the discussion informal.) Consider two processors that are asynchronous and do not wait for one another. A flip-flop is a Boolean variable that can be read by one processor and written by the other. Suppose, one is given atomic flip-flops as building blocks, and is asked to implement a k-bit atomic variable, that can be written by one processor and read by the other. Of course, a buffer consisting of k flip-flops suffices to hold such a value. If, however, the implementation allows the reader to read and return the value held by the same buffer that the writer may simultaneously access for writing, then either the writer or the reader might do all of its accesses while the other is temporarily stopped halfway by the buffer. The problem arises that a reader might obtain 1111000 from a buffer that a writer is changing from 0000000 to 1111111. That is, the reader would obtain a value consisting of half the new value and half the old one. Obviously, this violates atomicity. The problem then is to design a protocol that provides exclusive access to buffers without waiting. Correct implementations of atomic multi-bit variables from single bits can be found in Peterson [1983], Lamport [1986], Vidyasankar [1988], and Tromp [1989].

These atomic variables serve as the building blocks of our construction of an n-user variable; a variable shared between n users each of which can atomically execute both read and write operations.

At the outset, we state our main result:

THEOREM 1.1.1. *An atomic n-user variable is implemented wait-free from* $O(n^2)$ *atomic* 1*-reader* 1*-writer variables each with* $O(n)$ *control bits. Moreover, it uses* $O(n)$ *accesses per Read/Write running in* $O(1)$ *parallel time.*

Our notion of parallel time allows a set of accesses to different variables to proceed in arbitrary order in one time-unit.

1.2. COMPARISON WITH RELATED WORK. The first version of our construction was widely circulated in 1987 as the preprint [Li and Vitányi 1987], followed by a conference version [Li and Vitányi 1989], testing by implementation, and a preprint of the current version [Li et al., 1989]. Since 1987 the field has blossomed to such an extent that we cannot survey it in detail. Therefore, we

restrict ourselves to listing the related papers published before that time and the most relevant later work, and compare our results in detail only with the best existing ones we know of.

With this in mind, related constructions are given by Singh et al. [1994], Kirousis et al. [1989], Burns and Peterson [1987], Newman-Wolfe [1987], Israeli and Li [1987], and Haldar and Vidyasankar [1995] for the single-reader to multi-reader case, and by Vitányi and Awerbuch [1986], Peterson and Burns [1987], Schaffer [1988], and Israeli and Shaham [1992] for the multi-reader to multi-writer case. The latter problem, from multi-reader to multi-writer, is the more difficult one. It is by now well-known that these constructions are difficult and error-prone. Since it is by no means easy to find the errors, we point out that the bounded control bit solutions in Vitányi and Awerbuch [1986], Peterson and Burns [1987], and Burns and Peterson [1987] are known to be incorrect as noted in Peterson and Burns [1987], Schaffer [1988], and Haldar and Vidyasankar [1992], respectively. The latter two provide corrected constructions, while Vitányi and Awerbuch [1986] is fixed using the (largely identical) method of Dwork and Waarts [1992]. The *unbounded* solution in Vitányi and Awerbuch [1986] is correct and is used as a point of departure in the current paper.

Here, we present the first implementation of an n-user variable directly from single reader variables. The algorithm uses $O(n)$ accesses to single-reader variables per operation, and each single-reader variable stores two copies of the value of the constructed variable together with $O(n)$ bits of control information. Most other algorithms use multi-reader variables instead of single-reader ones, making a direct comparison impossible. Still, an indirect comparison can be made by combining an algorithm using multi-reader variables with an implementation of a multi-reader variable from single-reader ones.

The multi-writer algorithm in Schaffer [1988] uses $\Theta(n^2)$ accesses to multi-reader variables per operation, running in $\Theta(n)$ parallel time.

Israeli and Li's important paper [Israeli and Li 1987] introduced the notion of a *bounded time-stamp system*, which is a general mechanism for tracking the order of events in a system. They developed an elegant theory of *sequential time-stamp systems*, in which operations are totally ordered, as well as a system with a limited amount of concurrency. A proper definition of concurrent time-stamp system was introduced in Dolev and Shavit [1996]. Such a system can be applied directly to solve the problem of implementing a multi-user variable from multi-reader variables. Their technique yields an implementation using $\Theta(n \log n)$ accesses to (linear-sized) multi-reader variables per operation, running in $\Theta(\log n)$ parallel time.

Implementing the multi-reader variables in single-reader single-writer variables by known constructions, such as Singh et al. [1994], to obtain a multi-writer variable implementation from single-reader single-writer variables, multiplies the number of accesses per operation in the above constructions by a factor n.

More recent papers[1] provide improved constructions for concurrent time-stamp schemes. The most economic of these constructions [Israeli and Pinhasov 1992] uses $\Theta(n)$ accesses to (linear-sized) multi-reader variables per operation, running in $O(1)$ parallel time, and can be applied directly to the construction of

[1] See, for example, Dwork and Waarts [1992], Israeli and Pinhasov [1992], Gawlick et al. [1992], and Dwork et al. [1992].

a multi-writer variable. Compared to our construction, this still has the disadvantage of using multi-reader variables and having a higher conceptual complexity.

Bloom [1988] presented an elegant 2-writer construction. Herlihy [1988] considers more powerful shared objects that have no wait-free implementations from variables.

A more recent construction than ours [Israeli and Shaham 1992] presents a direct solution that is optimal in space (logarithmic control bit complexity, see Li and Vitányi [1992]) as well as the number of variable accesses per read/write (linear). They do not however achieve constant parallel time (to be defined in the next section) and have a rather more complicated protocol.

We believe the construction presented here is relatively simple and transparent. Both the problem of how to implement a multi-reader variable from single-reader variables, and the problem of implementing a multi-writer variable from multi-reader variables, are solved by simplifications of our main solution.

The basis of our proof-technique was developed in Awerbuch et al. [1988]. Our model and terminology is based on Herlihy and Wing [1990], which defines and motivates the notion of linearizability.

1.3. MULTI-USER VARIABLE CONSTRUCTION. In this section, we consider the problem of constructing an n-user variable from single-reader variables and state the correctness condition such a construction has to satisfy.

Throughout the paper, the n users are indexed with the set $I = \{0, \ldots, n-1\}$. The variable constructed will be called ABS (for abstract).

A construction consists of a collection of shared variables $R_{i,j}$, $i, j \in I$ (providing a communication path from user i to user j), and two procedures, *Read* and *Write*. Both procedures have an input parameter i, which is the index of the executing user, and in addition, Write takes a value to be written to ABS as input. An implicit or explicit return statement ends the execution of both procedures, in the case of Read having an argument that is taken to be the value read from ABS.

A procedure contains a declaration of local variables and a body. A local variable appearing in both procedures can be declared *static*, which means it retains its value between procedure invocations. The body is a program fragment comprised of atomic statements. Access to shared variables is naturally restricted to assignments from $R_{j,i}$ to local variables and assignments from local variables to $R_{i,j}$, for any j (recall that i is the index of the executing user). No other means of interprocess communication is allowed. In particular, no synchronization primitives can be used. Assignments to and from shared variables are called writes and reads respectively, always in lowercase.

The *space complexity* of a construction is the maximum size, in bits, of a shared variable.

The *time complexity* of the Read or Write procedure is the maximum number of shared variable accesses in a single execution.

A *parallel loop* is a loop denoted as '**for** $j \in I$' and it is parallel in the sense that its iterations[2] can be executed in arbitrary order. Moreover, shared variables accessed in different iterations of the parallel loop must be disjoint. The *parallel time complexity* of the Read or Write procedure differs from the normal,

[2] This is a slight abuse of the term, since the word iteration suggests sequential behavior.

sequential one in that the cost of a parallel loop is the maximum, rather than the sum, of the number of shared variable accesses in each of its iterations (we don't consider the case of nested parallel loops in this paper). The total cost of the procedure is then simply the sum of the costs of its parallel loops plus the maximum number of remaining shared variable accesses.

A construction must satisfy the following constraint.

Wait-Freedom. Each procedure must be free from unbounded loops.

Given a construction, we are interested in properties of its executions, which the following notions help formulate. A *state* is a configuration of the construction, comprising values of all shared and local variables, as well as program counters. Note that we need a somewhat liberal notion of program counter to characterize the execution of a parallel loop. In between invocations of the Read and Write procedure, a user is said to be idle, and its program counter has the value 'idle'. One state is designated as *initial state*. All users must be idle in this state.

A state t is an *immediate successor* of a state s if t can be reached from s through the execution of a procedure statement by some user in accordance with its program counter. Recall that n denotes the number of users of the constructed variable ABS. A state has at least n immediate successors: If a user is idle, it can invoke either the Read or Write procedure. And if it is within one of these procedures, there is at least one atomic statement to be executed next (possibly more during the execution of a parallel loop).

A *history* of the construction is a finite or infinite sequence of states t_0, t_1, t_2, ... such that t_0 is the initial state and t_{i+1} is an immediate successor of t_i. Transitions between successive states are called the *events* of a history. With each event is associated the index of the executing user, the relevant procedure statement, and the values manipulated by the execution of the statement. Each particular access to a shared variable is an event, and all such events are totally ordered.

The (*sequential*) *time complexity* of the Read or Write procedure is the maximum number of shared variable accesses in some such operation in some history. *Parallel time complexity* is defined similarly, except that for each parallel loop, we count not the sum of the time complexities of its iterations, but rather their maximum.

An event a *precedes* an event b in history h, $a \prec_h b$, if a occurs before b in h. The subscript h is omitted when clear from context. Call a finite set of events of a history an event-set. Then we similarly say that an event-set A precedes an event-set B in a history, $A \prec_h B$, when each event in A precedes all those in B. We use $a \preceq b$ to denote that either $a = b$ or $a \prec b$. The relation $\prec_h$ on event-sets constitutes what is known as an *interval order*. That is, a partial order satisfying the interval axiom $a \prec b \wedge c \prec d \wedge c \not\prec b \Rightarrow a \prec d$. This implication can be seen to hold by considering the last event of c and the earliest event of b. See Lamport [1986] for an extensive discussion on models of time.

Of particular interest are the sets consisting of all events of a single procedure invocation, which we call an *operation*. An operation is either a Read operation or a Write operation. It is *complete* if it includes the execution of the (possibly implicit) **return** statement of the procedure. Otherwise, it is said to be *pending*. A history is complete if all its operations are complete. Note that in the final state

of a complete finite history, all users are idle. The value of an operation is the value written to ABS in the case of a Write, or the value read from ABS in the case of a Read.

The following crucial definition expresses the idea that the operations in a history appear to take place instantaneously somewhere during their execution interval. A more general version of this is presented and motivated in Herlihy and Wing [1990]. To avoid special cases, we introduce the notion of a *proper* history as one that starts with an initializing Write operation that precedes all other operations.

Linearizability. A complete proper history h is *linearizable* if the partial order $\prec_h$ on the set of operations can be extended to a total order which obeys the semantics of a variable. That is, each Read operation returns the value written by that Write operation which last precedes it in the total order.

Definition 1.3.1. A construction is *correct* if it satisfies Wait-Freedom and all its complete proper histories are linearizable.

1.4. THE TAG FUNCTION. Although the definition of linearizability is quite clear, it is convenient to transform it into an equivalent specification from which the first algorithm can be directly derived. The idea behind the following lemma was first expressed by Lamport [1986, Proposition 3] for the case of a single writer. In Singh et al. [1994], the equivalent conditions given by Lamport's proposition are in fact taken as the definition of linearizability (often called atomicity in the register construction literature). The Atomicity Criterion of Awerbuch et al. [1988] is the first generalization of Lamport's proposition to the case of multiple readers and writers. A further generalization appears in Anderson [1993] for the case of a variable having several fields which can be written independently.

LEMMA 1.4.1. *A complete proper history h is linearizable iff there exists a function mapping each operation in h to a rational number, called its tag, such that the following* 3 *conditions are satisfied*:

Uniqueness. Different Write operations have different tags.

Integrity. For each Read operation there exists a Write operation with the same tag and value, that it doesn't precede.

Precedence. If one operation precedes another, then the tag of the latter is at least that of the former.

PROOF

$\Rightarrow$ Let a complete proper history h be linearizable. Let $<$ be the total order extending $\prec_h$ according to the definition of linearizability. Assign to each operation a tag which is the number of Write operations up to and including it in $<$. This clearly satisfies Uniqueness. For each Read operation R, the Write operation W that precedes it last in $<$ has the same tag. Also, because $<$ obeys the semantics of a variable, W and R have the same value. From the facts that $<$ extends $\prec_h$, $W < R$, and $<$ is acyclic, we conclude that $\neg R \prec_h W$. So Integrity is satisfied as well. Finally, for operations $A \prec_h B$, we necessarily have $A < B$ and thus the tag of B is at least that of A.

```
type I : 0..n − 1
     shared : record
                  value : ABStype
                  tag   : integer
              end
```

```
procedure Write(i, v)
var j : I
    t : integer
    from : array[0..n − 1] of shared
begin
    for j ∈ I do from[j] := R_{j,i}
    select t such that (∀j : t > from[j].tag) ∧ t ≡ i   (mod n)
    from[i] := (v, t)
    for j ∈ I do R_{i,j} := from[i]
end
```

```
procedure Read(i)
var j, max : I
    from : array[0..n − 1] of shared
begin
    for j ∈ I do from[j] := R_{j,i}
    select max such that ∀j : from[max].tag ≥ from[j].tag
    from[i] := from[max]
    for j ∈ I do R_{i,j} := from[i]
    return from[i].value
end
```

FIG. 1. Construction 0.

$\Leftarrow$ Suppose we are given a complete proper history h and a function *tag* satisfying the three conditions. Using Uniqueness, totally order the Write operations according to their tags. Next, we insert all Read operations in this total order: for each Write operation in turn, insert immediately after it those Read operations having the same tag, in some order extending $<_h$. By Integrity, the result is a total order $<$ on all operations, that obeys the semantics of a variable. It remains to show that $<$ extends $<_h$. Suppose $A <_h B$ are two operations. By Precedence, A's tag is at most that of B. If A's tag is less than B's, or A and B are Read operations with the same tag, then $A < B$ follows from the construction of $<$. In the remaining case, A and B have equal tags and at least one of them is a Write operation. By Uniqueness, one is a Read operation, and the other is the unique Write operation with the same tag. Finally, we use Integrity to conclude that A is the Write, and B the Read operation. Thus, $A < B$ follows again from the construction of $<$. □

2. *The Basic Unbounded Construction*

Figure 1 shows Construction 0, which is the unbounded solution of Vitányi and Awerbuch [1986]. We present it here as an aid in understanding Construction 1, and give only a sketchy proof.

The Write and Read procedures are given after the declaration of the type of the shared variables $R_{i,j}$. The initial state of the construction has all $R_{i,j}$ containing (0, 0).

The tag function called for in Lemma 1.4.1 is built right into this construction. Each operation starts by collecting value-tag pairs from all users. In the third line

of either procedure, the operation picks a value and tag for itself. It finishes after distributing this pair to all users. It is not hard to see that the three conditions of Lemma 1.4.1 are satisfied for each complete proper history. Integrity and Precedence are straightforward to check. Uniqueness follows since tags of Write operations of different users are not congruent modulo n, while tags of Write operations of a single user strictly increase (based on the observation that each $R_{i,j}.tag$ is nondecreasing).

3. *Solution Method*

The only problem with Construction 0 is that the number of tags is infinite. With a finite number of tags comes the necessity to reuse tags and hence to distinguish old tags from new ones.

In Construction 1, we introduce a shooting mechanism to provide additional aging information to the tags. At the start of an operation, a user sets up a fresh "target" that gets "shot at" by Write operations. A tag issued by an operation is considered old once its associated target has received sufficiently many shots. The shooting mechanism also serves another purpose, which is that of approximating a snapshot, an instantaneous picture of the state of a set of shared variables. In Construction 0, an operation collects information on values and tags of all users by reading their shared variables one after another, in arbitrary order (the first line in either procedure). Since these read events are interleaved with events of other users, in particular write events, the picture it obtains this way may be very distorted. In Construction 1, with additional information to collect, there is a need to limit the amount of distortion.

If, after the information-collecting period, the initially fresh target has received too many shots, then the operation will *abort*, that is, terminate without executing the remainder of the procedure. Aborting operations do not change or otherwise make use of any tags and thus have very limited interaction with non-aborting operations. The latter in turn obtain a good, if not instantaneous, picture of the shared state. In fact, the picture is good enough to enable them to discriminate old tags by inspection of the associated targets. This discrimination feature is however not yet implemented in Construction 1. While it compares unbounded tags as in Construction 0, it employs many additional unbounded counters that we can prove certain properties about, showing that all unboundedness is redundant. Thus, Construction 1 paves the way to our final, bounded, solution.

In Section 3.1, we discuss Construction 1 and in particular the shooting mechanism, in more detail. Section 3.2 introduces some notational conventions. The correctness proof is given in Section 3.3. Finally, Section 4 shows how Construction 1 can be changed into an equivalent one using only bounded counters.

3.1. CONSTRUCTION 1. Figure 2 shows the data-structure and procedures of Construction 1. The Write procedure turns out to be an extension of the Read procedure, which is why the two are more conveniently shown together. The line indicated '(Read only)' is unique to the Read procedure, making the remaining lines effectively unique to the Write procedure. The initial state of Construction 1 has 0 in all fields of all shared and static variables.

Let's look at the data structures used in the construction. The *value* and *tag* fields have exactly the same function as in Construction 0. The *prev* field is used

```
type I : 0..n − 1
     shared : record
                 value,prev : ABStype
                 tag  : integer
                 ss : 0..1
                 shoot,heal : array[0..1][0..n − 1] of integer
              end
```

```
procedure Read(i) / Write(i,v)
var j : I
    t : integer
    s : 0..1
    from,tmp : array[0..n − 1] of shared
    static me : shared
begin
    s := 1 − me.ss
s:  for j ∈ I do me.heal[s][j] := R_{j,i}.shoot[s][i]
h:  for j ∈ I do R_{i,j} := me
r:  for j ∈ I do from[j] := R_{j,i}
t:  for j ∈ I do tmp[j] := R_{j,i}
    if ∃j ∈ I : tmp[j].shoot[s][i] − me.heal[s][j] ≥ 3
    then return tmp[j].prev
    select max such that ∀j : from[max].tag ≥ from[j].tag
    me.prev, me.value, me.tag, me.ss :=
          me.value, from[max].value, from[max].tag, s
p:  for j ∈ I do R_{i,j} := me
    (Read only) return me.value
    for j ∈ I, s ∈ {0..1} do
        if me.shoot[s][j] − from[j].heal[s][i] < 6
        then me.shoot[s][j]+ := 1
    select t such that t − me.tag ∈ {1,...,n} ∧ t ≡ i  (mod n)
    me.value, me.tag := v, t
w:  for j ∈ I do R_{i,j} := me
end
```

FIG. 2. Construction 1.

to remember values of former operations, which are used by aborting Read operations. Two sets of heal counters, $heal[0][0\ ..\ n-1]$ and $heal[1][0\ ..\ n-1]$, are used to hold targets. The *ss* (shoot-selector) field selects which of the two sets holds the target associated with the current value-tag pair. A second set is needed since new operations must set up a target before they can compute a new tag. Together with the heal counters, the shot counters, $shoot[0\ ..\ 1][0\ ..\ n-1]$, implement the shooting mechanism. User j shoots at a target $heal[s][0\ ..\ n-1]$ of user k by making his counter $shoot[s][k]$ larger than the counter $heal[s][j]$ of user k, up to a maximum of 6.

Consider the procedures. The lines involving shared variable access are identified by one of the characters s, h, r, t, p, and w, which are mnemonic shorthands for setup, heal, read, test, propagate and write, respectively.

At the start of an operation, say a, user i sets up a new target in the available heal counter set $(1 - me.ss)$ by catching up with each user's shot counter. It then writes out the target in line h so that the other users can start shooting it. After collecting every one's data in line r, it proceeds to test in line t how many times its target has been shot. More precisely, if some user has increased its shot counter at least three times since it was previously read in line s, then a will abort. For the sake of definiteness, let the j in the **return** statement be the

minimal index satisfying the condition of the test. It can be shown that a completely "contains" an operation b of user j with the value $tmp[j].prev$, as well as a write operation w of user j that precedes or equals b. Thus, a can be imagined to have occurred right before w or after b in a linearization, depending on whether a is a Write or Read operation. If no user shot the target three times, then user i sets *max* to an index of the largest visible tag. It then saves the old value in *prev*, changes its value and tag to that of *max*, and associates its target with the new value-tag pair. In line p, record *me* is written out. The purpose of the Write operations propagating the value-tag pair of *max* is to ensure that before any user can see the Write's new tag, all users will be able to see a tag (the propagated one) which is at most n smaller. This fact will be used in Lemma 4.1.1 to show that outdated tags are easily recognized. The Read procedure ends after line p by returning the value copied from *max*.

The Write procedure continues by shooting all visible targets, that is, increasing all its shot counters that are not already six ahead of their corresponding heal counter. User i next chooses a tag unique to it which is larger than all visible ones. This is paired with v, the argument of the Write procedure, and all is written out in line w.

3.2. NOTATIONAL CONVENTIONS. The following notions are used in the proof. Assume an arbitrary but fixed history. The mth nonaborting operation of user i is denoted N_i^m. If $a = N_i^m$ then a^{+r} denotes N_i^{m+r}, that is, the rth next nonaborting operation by user i following a, assuming it exists. If $a = N_i^m$, then a^{-r} denotes N_i^{m-r}, that is, the rth previous nonaborting operation by user i preceding a. Use of this notation depends on the assumption that $r < m$. Since all shot counters are initialized to 0, and increase at most by one per nonaborting Write operation, the value assigned by an operation a to one of its shot counters provides a lower bound on m. We'll use the notation only where it is justified on these grounds.

The events of an operation a involving shared variable access constitute up to six events-sets, or *phases*:

$$a.\mathrm{s} < a.\mathrm{h} < a.\mathrm{r} < a.\mathrm{t} < a.\mathrm{p} < a.\mathrm{w},$$

in accordance with the labeled lines of the Read and Write procedure. Aborting operations consist of only the first four phases, while a nonaborting Read operation has the first five. The n events in a phase $a.\mathrm{c}$ (c one of s, h, r, t, p, or w) are denoted $a.\mathrm{c}_j$ with $j \in I$, and are called c-*events*.

For a shared variable read event e, define p-*Last*(e) to be the operation containing the last p-event preceding e that accesses the same shared variable. If such an event does not exist, then p-*Last*(e) is defined to be the nonoperation $\perp$.

For a an operation and *exp* an expression consisting of (symbolic or explicit) constants and local variables, define *exp*@a as the final value of that expression in the procedure invocation corresponding to a. Array indices i, j, k, s, t refer to symbolic constants defined in the context, not to the local variables. Define

$$value@\perp = prev@\perp = tag@\perp = ss@\perp = shoot[\cdot][\cdot]@\perp = heal[\cdot][\cdot]@\perp = 0,$$

in accordance with the initialization of the construction. Define $exp@a.\mathrm{c}$ (c one of s, h, r, t, p, or w) as the value of the expression *exp* after completion of line c

of the procedure invocation corresponding to a. By convention, the prefix *me.* is omitted when *exp* is a field of *me.*

3.3. CORRECTNESS OF CONSTRUCTION 1. Construction 1 trivially satisfies Wait-Freedom since all loops range over $I = \{0, \ldots, n-1\}$. Therefore, we only need to prove correctness. That is, we must show that each complete proper history is linearizable. By Lemma 1.4.1, this means we need to prove uniqueness, integrity, and precedence. First, we need some preparatory claims.

CLAIM 3.3.1. *All shared tag, heal and shot counters are nondecreasing in the course of a history.*

PROOF. A shared variable $R_{i,j}$ is changed only when *me* is written to it, in an h-, p-, or w-event of user i, so a nondecreasing counter in the static local variable *me* of user i implies a corresponding nondecreasing counter in $R_{i,j}$ for all $j \in I$. The *me.shoot* counters are only incremented and therefore nondecreasing. Hence, so are the shared shot counters $R_{i,j}$. Each heal counter *me.heal*[s][j] of user i is only changed by assignment from $R_{j,i}$.*shoot*[s][i] and is thus also nondecreasing. It remains to show that *me.tag* is nondecreasing. Consider the new tag *from*[*max*].*tag* that is assigned to *me.tag* prior to line p. By the selection of *max*, this is at least *from*[i].*tag* which by lines h and r is just a copy of *me.tag*. Thus, *me.tag* doesn't decrease in this assignment. In the other assignment, prior to line w, *me.tag* only increases. □

COROLLARY 3.3.2. *If event e writes v_e to a shared tag, heal, or shot counter, and event f reads v_f from the same shared counter, then $e \prec f \Rightarrow v_f \geq v_e$ and $v_f < v_e \Rightarrow f \prec e$.*

COROLLARY 3.3.3. *Let $a \prec b$ be nonaborting operations by users i and j, respectively. If b is a Read operation, then*

$$tag@b \geq tag@a,$$

and if b is a Write operation, then

$$tag@b \geq tag@a + 1.$$

PROOF. By the selection of *max*, $a \prec b$, and corollary 3.3.2, *from*[*max*].*tag*@b.p $\geq$ *from*[i].*tag*@b.r $\geq$ *tag*@a. For b a nonaborting Read operation, *tag*@b = *from*[*max*].*tag*@b.p. For b a nonaborting Write operation, *tag*@$b \geq$ *tag*@b.p + 1 = *from*[*max*].*tag*@b.p + 1. □

CLAIM 3.3.4. *The differences tmp*[j].*shoot*[s][i] − *me.heal*[s][j] *and me.shoot*[s][j] − *from*[j].*heal*[s][i] *between corresponding shot and heal counters as computed in line* t + 1 *and line* p + 3 *are between* 0 *and* 6 *(inclusive).*

PROOF. For each fixed i, j and s, $0 \leq R_{j,i}$.*shoot*[s][i] − $R_{i,j}$.*heal*[s][j] ≤ 6 is an invariant. It holds initially because of zero initialization. According to Claim 3.3.1, this invariant can be violated only if either user i assigns a value larger than $R_{j,i}$.*shoot*[s][i] to $R_{i,j}$.*heal*[s][j] in line h, or if user j writes a value larger than $R_{i,j}$.*heal*[s][j] + 6 to $R_{j,i}$.*shoot*[s][i] in line w. But user i only makes indirect copies from $R_{j,i}$.*shoot*[s][i] (read in line s) to $R_{i,j}$.*heal*[s][j]. And user j only increments $R_{j,i}$.*shoot*[s][i] (line p + 4) after seeing *from*[j].*heal*[s][j] $\geq$

$R_{j,i}.shoot[s][i] - 5$ (line p + 3), in which the first term is at most $R_{i,j}.heal[s][j]$. Thus, both cases give a contradiction and the invariant holds. This implies the same bounds on the computed differences. □

CLAIM 3.3.5. *Let a be an operation by user i, and j be some index. Let either* $b = \text{p-}Last(a.\text{r}_j)$ *and* $X = from[j]@a.\text{r}_j$, *or* $b = \text{p-}Last(a.\text{t}_j)$ *and* $X = tmp[j]@a.\text{t}_j$. *Let* $s = X.ss$. *Then*

(1) $X.prev = prev@b$, $s = ss@b$

(2) *if* $b = \perp$ *then* $X.tag = 0$
else $tag@b.\text{p} \leq X.tag \leq tag@b$

(3) *for all* $k \in I$, $X.heal[s][k] = heal[s][k]@b$

(4) *for all* $k \in I$, $z \in \{0, 1\}$, *if* $b = \perp$ *then* $X.shoot[z][k] = 0$
else $shoot[z][k]@b.\text{p} \leq X.shoot[z][k] \leq shoot[z][k]@b$

PROOF

(1) Only the p-events of user j change $R_{j,i}.prev$ and $R_{j,i}.ss$.

(2) In case $b = \perp$, no tag has overwritten the initial 0. In case $b \neq \perp$, the first inequality follows directly from the definition of b and Corollary 3.3.2. For the second, note that after b, the value of $R_{j,i}.tag$ remains $tag@b$ until $b^{+1}.\text{p}_i$ (if any), which by definition of b doesn't precede the reading of X.

(3) After $b.\text{p}_i$ (or from the start of history in case $b = \perp$), the value of ss in user j's *me* remains s at least until line p − 1 of b^{+1} (if it exists). Hence, its $heal[s]$ counters remain unchanged until the next operation after b^{+1}.

(4) Analogous to item (2). □

CLAIM 3.3.6. *Let* a, $b = a^{+1}$ *be two nonaborting operations by user i.*

(1) $prev@b = value@a$

(2) $\forall j \in I, s \in \{0, 1\} : shoot[s][j]@b \leq shoot[s][j]@a + 1$

(3) *If a and b are Write operations, then* $tag@b \geq tag@a + n$.

PROOF

(1) Since b doesn't abort, and aborting operations don't change *me.value*, $prev@b = value@b.\text{s} = value@a$.

(2) Similarly.

(3) Since $tag@b \equiv tag@a \equiv i \pmod{n}$, their difference is a multiple of n, and by Corollary 3.3.3, it is positive. □

CLAIM 3.3.7. *Let a be an aborting operation by user i, and let j be the minimal index for which the abortion condition holds. Then there exists a nonaborting Write operation w and a nonaborting operation b by user j, such that*

$$a.\text{s}_j \prec w \preceq b \prec a.\text{t}_j \wedge tmp[j].prev@a = value@b.$$

PROOF. Let $c = \text{p-}Last(a.\text{t}_j)$, let $b = c^{-1}$, let w be the last Write among ... b^{-2}, b^{-1}, b and let $d = w^{-1}$ (recall Section 3.2 on notation). Then, $d \prec w \preceq b \prec c.\text{p} \prec a.\text{t}_j$ and by Claims 3.3.5 and 3.3.6, $tmp[j].prev@a = prev@c = value@b$. Also, with $s = 1 - ss@a$, by (respectively) abortion of a and Claim 3.3.5 and Claim 3.3.6, and definition of w, and Claim 3.3.6,

$$heal[s][j]@a + 3 \leq tmp[j].shoot[s][i]@a \leq shoot[s][i]@c$$

$$\leq shoot[s][i]@b + 1 = shoot[s][i]@w + 1 \leq shoot[s][i]@d + 2.$$

This shows that $heal[s][j]@a < shoot[s][i]@d$; hence, not $d < a.s_j$. Combined with $d < w$ this yields $a.s_j < w$. □

The following claim will be used in later sections.

CLAIM 3.3.8. *If a is an operation by user i, and w_1, w_2, w_3 are nonaborting Write operations by user k, such that*

$$a.h_k < w_1.r_i < w_2 < w_3.w_i < a.t_k,$$

then a aborts.

PROOF. Let $s = 1 - ss@a.t$. Claim 3.3.4 shows that $me.shoot[s][i]@w_1.r - from[i].heal[s][k] \geq 0$ in line p + 3 of w_1. Since these values do not change in between, this also holds in line r of w_1. This and the assumption of the claim give

$$shoot[s][i]@w_1.r \geq from[i].heal[s][k]@w_1.r = heal[s][k]@a.t.$$

According to the shooting mechanism, induction on m shows that $shoot[s][i]@w_m \geq heal[s][k]@a.t + \min(m, 6)$. Since $w_3.w_i < a.t_k$, Corollary 3.3.2 implies

$$tmp[k].shoot[s][i]@a.t \geq shoot[s][i]@w_3 \geq heal[s][k]@a.t + 3,$$

hence a aborts. □

LEMMA 3.3.9. *Each complete proper history h of Construction 1 is linearizable.*

PROOF. The proof is based on the tag lemma. We show that there is a function $\tau(\ )$, mapping each operation in h to a rational number, that satisfies Uniqueness, Integrity, and Precedence. Let a be an operation by user i. If a doesn't abort, then simply set $\tau(a) = tag@a$. Otherwise, if a aborts, let b be the operation given by Claim 3.3.7. Now set $\tau(a) = tag@b$ if a is a Read operation, or set $\tau(a) = tag@b - \epsilon_a$, if a is a Write operation, where $0 < \epsilon_a < 1$ is a fraction unique to a.

Uniqueness. Let a and b be different Write operations by users i and j, respectively. If either aborts, then its tag has a unique fractional part and is therefore different from the other operation's tag. Suppose neither aborts. Then $\tau(a) = tag@a \equiv i \pmod n$, and $\tau(b) = tag@b \equiv j \pmod n$. If $i \neq j$, then Uniqueness follows immediately. In case $i = j$, one Write operation must precede the other, and Uniqueness follows from Corollary 3.3.3.

Integrity. For aborting Read operations, Integrity follows from Claim 3.3.7. The value-tag pair that a nonaborting Read operation a copies must originate from a nonaborting Write operation b. Clearly, $\neg(a < b)$. Combined with the definition of τ, this proves Integrity.

Precedence. Consider two operations $a < b$. We must show that $\tau(a) \leq \tau(b)$. If a aborts, then by Claim 3.3.7 and definition of τ, there exists a j and a nonaborting operation a' such that $a' < a.t_j < b$ and $\tau(a) \leq \tau(a')$, in which case it would suffice to show Precedence for $a' < b$. So without loss of

generality we can assume that a doesn't abort. If b doesn't abort, then Precedence follows from Corollary 3.3.3.

Suppose b aborts. By Claim 3.3.7, there exists a nonaborting Write operation w and a nonaborting operation b' by some user j, such that $a \prec b.\mathrm{s}_j \prec w \preceq b'$. By Corollary 3.3.3, $tag@b' \geq tag@w \geq tag@a + 1 = \tau(a) + 1$. Since $\tau(b)$ is either $tag@b' - \epsilon_b$ or $tag@b'$, $\tau(b) \geq \tau(a)$ follows. □

4. *Bounding the Counters*

Having proven Construction 1 correct, we will make a correctness preserving transformation that renders all variables essentially bounded, that is, that subsequently allows us to replace them with bounded versions. The transformation is based on three key lemmas. The first formalizes the idea that a tag, whose target is seen to have been shot sufficiently many times, can be considered old, and ignored in the selection of a maximum tag. The second shows that the remaining, "live", tags are in a bounded range, which is the basis for bounding the tags. Finally, the third shows that the perceived number of times a target is shot is bounded both from below and above, which is the basis for bounding the heal and shot counters.

4.1. Old Tags

LEMMA 4.1.1. *Let a be a nonaborting operation by user i. Let $j, k \in I$, and $s = from[j].ss@a.\mathrm{r}$. If*

$$from[k].shoot[s][j]@a.\mathrm{r} - from[j].heal[s][k]@a.\mathrm{r} \geq 6$$

(a sees 6 shots by k on j's target), then

$$from[k].tag@a.\mathrm{r} > from[j].tag@a.\mathrm{r}.$$

PROOF. Let $b = \text{p-}Last(a.\mathrm{r}_j)$ and $c = \text{p-}Last(a.\mathrm{r}_k)$. Using in succession Claim 3.3.5, the assumption of the claim, and Claim 3.3.5 again:

$$shoot[s][j]@c \geq from[k].shoot[s][j]@a.\mathrm{r}$$

$$\geq from[j].heal[s][k]@a.\mathrm{r} + 6 = heal[s][k]@b + 6$$

This shows the existence of

$$w_3 \prec w_4 \prec w_5 \prec w_6.\mathrm{w}_i \prec a.\mathrm{r}_k,$$

where w_m, $3 \leq m \leq 6$ is the *first* nonaborting Write operation by user k such that

$$shoot[s][j]@w_m = heal[s][k]@b + m.$$

By Corollary 3.3.2, Claim 3.3.6, and the tag choice in Construction 1,

$$from[k].tag@a.\mathrm{r} \geq tag@w_6 \geq tag@w_4 + 2n$$

$$\geq from[max].tag@w_4.\mathrm{p} + 2n + 1.$$

If $b = \perp$, then $from[j].tag@a.\mathrm{r} = 0$ and the lemma follows immediately. Otherwise, by Claim 3.3.5 and the tag choice in Construction 1,

$$from[j].tag@a.\mathrm{r} \leq tag@b \leq from[max].tag@b.\mathrm{p} + n.$$

Thus, to prove the lemma it suffices to show that

$$from[max].tag@w_4.\mathrm{p} + 2n + 1 > from[max].tag@b.\mathrm{p} + n. \qquad (1)$$

Claim 3.3.5 shows that $s = ss@b = 1 - ss@b.\mathrm{t}$. Since b doesn't abort and by definition of w_3,

$$tmp[k].shoot[s][j]@b.\mathrm{t}_k < heal[s][k]@b + 3 = shoot[s][j]@w_3.\mathrm{w}_j.$$

Hence, by Corollary 3.3.2, $b.\mathrm{r} < b.\mathrm{t}_k < w_3.\mathrm{w}_j < w_4$. Let Write operation w be the originator of the tag $from[max].tag@b.\mathrm{p}$. We have $\neg (b.\mathrm{r} < w.\mathrm{w})$. Since $w.\mathrm{p} < w.\mathrm{w}$ and $b.\mathrm{r} < w_4$, it must therefore be that $w.\mathrm{p} < w_4$. This shows

$$from[max].tag@w_4.\mathrm{p} \geq tag@w.\mathrm{p} \geq tag@w - n = from[max].tag@b - n,$$

which immediately implies (1). □

Definition 4.1.2. In the context of a Read or Write procedure, define

$$alive(j) \equiv (\forall k \in I : from[k].shoot[s][j] - from[j].heal[s][k] < 6),$$
$$\text{where } s = from[j].ss$$

COROLLARY 4.1.3. *For each nonaborting operation a, $alive(max)@a.\mathrm{p}$.*

This shows that the choice of *max* can be restricted to those $j \in I$ for which $alive(j)$ holds.

4.2. RANGE OF ALIVE TAGS. The parameter m in the next lemma serves to prepare for a later simplification of the construction, for the case of only one writer.

LEMMA 4.2.1. *Let a be a nonaborting operation by user i. Let $j \in I$ and $s = from[j].ss@a.\mathrm{r}$. Let $1 \leq m \leq n$ be the number of users with Write operations. If $alive(j)@a.\mathrm{r}$, then*

$$from[max].tag@a.\mathrm{p} - from[j].tag@a.\mathrm{p} \leq 10mn.$$

PROOF. Assume, to the contrary, that $from[max].tag@a.\mathrm{p} > from[j].tag@a.\mathrm{p} + 10mn$. Let W be the set of all nonaborting Write operations w such that $tag@w > from[j].tag@a.\mathrm{p} \wedge \neg (a.\mathrm{r} < w)$. Since a reads a tag that is more than $10mn$ greater than j's, and new tags are chosen in increments of at most n, we must have $|W| > 10m$. Therefore, some user, say k, has at least 11 operations, say $w_0 < w_1 < \cdots < w_{10}$, in W.

Let $b = \text{p-}Last(a.\mathrm{r}_j)$. We claim that

$$from[j].heal[s][k]@w_1 \geq heal[s][k]@b. \qquad (2)$$

Otherwise, Corollary 3.3.2 gives $b \neq \perp$ and $w_0 < w_1.\mathrm{r}_j < b.\mathrm{h}_k < b.\mathrm{r}_k$, from which Corollary 3.3.2 and Claim 3.3.5 imply $tag@w_0 \leq from[k].tag@b.\mathrm{r} \leq$

$tag@b.\mathrm{p} \le from[j].tag@a.\mathrm{p}$, contradictory to the definition of $w_0 \in W$. As the proof of Claim 3.3.8 shows, (2) implies $shoot[s][j]@w_6 \ge heal[s][k]@b + 6$. On the other hand, the assumption $alive(j)@a.\mathrm{r}$ and Claim 3.3.5 give $from[k].shoot[s][j]@a.\mathrm{r} < from[j].heal[s][k]@a.\mathrm{r} + 6 = heal[s][k]@b + 6$. Together, using Corollary 3.3.2, these inequalities show that

$$a.\mathrm{h}_k \prec a.\mathrm{r}_k \prec w_6.\mathrm{w}_i \prec w_7.\mathrm{r}_i.$$

If $w_9.\mathrm{w}_i \prec a.\mathrm{t}_k$, then Claim 3.3.8 would show that a aborts; hence, we must instead have

$$a.\mathrm{r} \prec a.\mathrm{t}_k \prec w_9.\mathrm{w}_i \prec w_{10},$$

in contradiction to the definition of $w_{10} \in W$. □

The lemma shows that all alive tags are from $10mn$ to 0 less than the maximum. The following is an easy consequence:

COROLLARY 4.2.2. *Let a be a nonaborting operation by user i. Let $1 \le m \le n$ be the number of users with Write operations. Let $j, k \in I$ such that $alive(j)@a.\mathrm{r} \wedge alive(k)@a.\mathrm{r}$. Then*

$$-10mn \le from[k].tag - from[j].tag \le 10mn.$$

4.3. BOUNDS ON PERCEIVED SHOTS

LEMMA 4.3.1. *Let a be a nonaborting operation by user i. Let $j, k \in I$, and $s = from[j].ss@a.\mathrm{r}$. Then*

$$-4 \le from[k].shoot[s][j]@a.\mathrm{r} - from[j].heal[s][k]@a.\mathrm{r} \le 9.$$

PROOF. Assume, to the contrary, that the difference is outside $\{-4, \ldots, 9\}$. We will reach the required contradiction by showing that the conditions of Claim 3.3.8 hold, implying that a aborts.

Let $b = \text{p-}Last(a.\mathrm{r}_j)$ and $c = \text{p-}Last(a.\mathrm{r}_k)$. In the first case, assume the difference is under -4. Then, by Claim 3.3.5, $s = ss@b$ and

$$heal[s][k]@b = from[j].heal[s][k]@a.\mathrm{r} \ge from[k].shoot[s][j]@a.\mathrm{r} + 5.$$

So $b \ne \perp$ and $b.\mathrm{s}_k$ read a shot counter from user k that's at least five greater than what a read in $a.\mathrm{r}_k$. This shows the existence of

$$a.\mathrm{r}_k \prec w_1.\mathrm{w}_i \prec w_2 \prec w_3 \prec w_4 \prec w_5.\mathrm{w}_j \prec b.\mathrm{s}_k,$$

where w_m, $1 \le m \le 5$ is the *first* nonaborting write action by user k such that

$$shoot[s][j]@w_m = from[k].shoot[s][j]@a.\mathrm{r} + m.$$

Consequently, the conditions of Claim 3.3.8 hold:

$$a.\mathrm{h}_k \prec a.\mathrm{r}_k \prec w_2 \prec w_4 \prec b.\mathrm{s}_k \prec b.\mathrm{p}_i \prec a.\mathrm{r}_j \prec a.\mathrm{t}_k.$$

This proves the first inequality of the lemma.

In the other case, assume the difference is over 9. Then, by Claim 3.3.5,

$$shoot[s][j]@c \geq from[k].shoot[s][j]@a.\mathrm{r}$$

$$\geq from[j].heal[s][k]@a.\mathrm{r} + 10 = heal[s][k]@b + 10.$$

This shows the existence of

$$w_7 \prec w_8 \prec w_9 \prec w_{10}.\mathrm{w}_i \prec a.\mathrm{r}_k,$$

where w_m, $7 \leq m \leq 10$ is the *first* nonaborting Write operation by user k such that

$$shoot[s][j]@w_m = heal[s][k]@b + m.$$

Claim 3.3.4 shows that

$$from[j].heal[s][k]@w_7.\mathrm{r} \geq shoot[s][j]@w_7 - 6 = heal[s][k]@b + 1. \qquad (3)$$

By definition of b, $a.\mathrm{r}_j \prec b^{+1}.\mathrm{p}_i$ (taking b^{+1} to be N_j^1 in case $b = \perp$). Equation (3) shows that b^{+1} must exist, and that $b^{+1} \prec w_7.\mathrm{r}_j$, since after $b.\mathrm{p}_i$, the value of ss in user j's *me* remains s at least until line $\mathrm{p} - 1$ of b^{+1} and its $heal[s]$ counters remain unchanged until the next operation after b^{+1}. Thus

$$a.\mathrm{h}_k \prec a.\mathrm{r}_j \prec b^{+1}.\mathrm{p}_i \prec w_7.\mathrm{r}_j \prec w_8 \prec w_{10}.\mathrm{w}_i \prec a.\mathrm{r}_k \prec a.\mathrm{t}_k.$$

Again the conditions of Claim 3.3.8 hold. This proves the second inequality of the lemma. □

4.4. Equivalence with Bounded Counters. For notational convenience, we introduce three binary operators $\ominus$, $\oplus$, and $\odot$:

Definition 4.4.1. For each pair of integers a, b, we have $a \ominus b$, $a \oplus b$, and $a \odot b$ uniquely defined by the equations

$$-4 \leq a \ominus b < 10 \wedge a \ominus b \equiv a - b \pmod{14}$$

$$-4 \leq a \oplus b < 10 \wedge a \oplus b \equiv a + b \pmod{14}$$

$$-10n^2 \leq a \odot b < 10n^2 + n \wedge a \odot b \equiv a - b \pmod{20n^2 + n}.$$

Let Construction 2 be the result of replacing the selection of *max* in Construction 1

$$\text{select } max \text{ such that } \forall j : from[max].tag \geq from[j].tag$$

with

$$\text{select } max \in A \text{ such that } \forall j \in A : from[max].tag \odot from[j].tag \geq 0$$

where $A = \{j \in I : \forall k \in I : from[k].shoot[z][j] \ominus from[j].heal[z][k] < 6$ where $z = from[j].ss\}$
and of replacing the subtractions in line t + 1 and line p + 3 of Construction 1 with the $\ominus$ operation.

Lemma 4.4.2. *Each history of Construction* 2 *is a history of Construction* 1.

```
type I : 0..n − 1
     tagtype : −10n²..10n² + n − 1
     shottype : −4..9
     shared : record
                value,prev : ABStype
                tag : tagtype
                ss : 0..1
                shoot,heal : array[0..1][0..n − 1] of shottype
              end
```

```
procedure Read(i) / Write(i, v)
var j : I
    t : tagtype
    s : 0..1
    from,tmp : array[0..n − 1] of shared
    static me : shared
begin
    s := 1 − me.ss
    for j ∈ I do me.heal[s][j] := R_{j,i}.shoot[s][i]
    for j ∈ I do R_{i,j} := me
    for j ∈ I do from[j] := R_{j,i}
    for j ∈ I do tmp[j] := R_{j,i}
    if ∃j ∈ I : tmp[j].shoot[s][i] ⊖ me.heal[s][j] ≥ 3
    then return tmp[j].prev
    select max ∈ A such that ∀j ∈ A : from[max].tag ⊙ from[j].tag ≥ 0
    where A = {j ∈ I : ∀k ∈ I : from[k].shoot[z][j] ⊖ from[j].heal[z][k] < 6
                        where z = from[j].ss}
    me.prev, me.value, me.tag, me.ss :=
      me.value, from[max].value, from[max].tag, s
    for j ∈ I do R_{i,j} := me
    (Read only) return me.value
    for j ∈ I, s ∈ {0..1} do
        if me.shoot[s][j] ⊖ from[j].heal[s][i] < 6
        then me.shoot[s][j]⊕ := 1
    select t ∈ tagtype such that t ⊙ me.tag ∈ {1, ..., n} ∧ t ≡ i  (mod n)
    me.value, me.tag := v, t
    for j ∈ I do R_{i,j} := me
end
```

FIG. 3. Construction 3.

PROOF. By Corollary 4.1.3 and a simple reordering of terms, the selection of *max* in Construction 1 gives, in each reachable state, the same new state as

$$\text{select } max \in A \text{ such that } \forall j \in A : from[max].tag - from[j].tag \geq 0$$

where $A = \{j \in I : alive(j)\}$;
which in turn gives, by Corollary 4.2.2 and Lemma 4.3.1, in each reachable state, the same new state as the selection of *max* in Construction 2. By Claim 3.3.4, the subtractions in line t + 1 and line p + 3 of Construction 1 give, in each reachable state, the same new state as the $\ominus$ operations in Construction 2. □

COROLLARY 4.4.3. *Each complete proper history of Construction 2 is linearizable.*

We next consider our bounded solution, Construction 3, shown in figure 3. It is identical to Construction 2 except for the type of tags and shot/heal counters, and the way they are increased. Note that in the 4th to last line, the selection of t is possible (and unique) because $20n^2 + n$ is a multiple of n. The initial state of

Construction 3 has 0 in all fields of all shared and static variables, like Constructions 1 and 2.

We prove Construction 3 correct using a proof method known as *forward simulation* (see the excellent overview article [Lynch and Vaandrager 1995]). Formally, a forward simulation from A to B is a relation f over the states of A and the states of B that satisfies:

(1) Each initial state of A is related to an initital state of B.
(2) If A has a transition from state s to state s' by action a and s is related to state t of B, then B has a transition by the same action a from state t to a state t' related to s'.[3]

As Lynch and Vaandrager [1995] show, the existence of such a relation implies that the histories of A are indistinguishable from those of B—each (finite or infinite) sequence of actions that A can take can also be taken by B. Our forward simulation from Construction 3 to Construction 2 uses the following equivalence relation:

Definition 4.4.4. A state s of Construction 3 is said to be *equivalent* to a state u of Construction 2 if they are identical up to *congruence* of tags and shot/heal counters, as follows. If tag_3 is the value of some local or shared variable of type *tagtype* in state s, and tag_2 is the value of that same variable in state u, then we require $tag_3 \equiv tag_2 \pmod{20n^2 + n}$. If $shot_3$ is the value of some local or shared variable of type *shottype* in state s, and $shot_2$ is the value of that same variable in state u, then we require $shot_3 \equiv shot_2 \pmod{14}$. All other variables and program counters in s and u must be identical.

THEOREM 4.4.5. *Each complete proper history of Construction* 3 *is linearizable.*

PROOF. We show that the equivalence relation defined above satisfies the requirement of the forward simulation. It will then follow that each history of Construction 3 is indistinguishable from a history of Construction 2. Since all complete proper histories of Construction 2 are linearizable, and since linearizability depends only on the precedence relation among the Read and Write operations plus the values the Reads return, all complete proper histories of Construction 3 are then also necessarily linearizable.

As to item 1, both constructions have a unique starting state in which all variables are initialized to 0, hence these two states are equivalent.

Now, suppose Construction 3 has a transition from a state s to a state s' by action a, and let s be equivalent to state u of Construction 2. We consider all possibilities for the statement that action a corresponds to:

(1) If a is some statement that doesn't involve tag, heal, or shoot counters, then Construction 2 has a transition with the same action a to to a state u' equivalent to s'.
(2) If a is some statement that copies a tag, heal, or shoot counter, then also Construction 2, which has identical copy statements, has a transition with the same action a to to a state u' equivalent to s'.

[3] This definition is slightly simplified from Lynch and Vaandrager [1995], where B can make additional 'internal action' transitions, which we needn't consider.

(3) If a is the statement that tests if $\exists j \in I : tmp[j].shoot[s][i] \ominus me.heal[s][j] \geq 3$, then the left-hand-side of the inequality is, by definition of $\ominus$, the same in the equivalent states s and u. Hence, Construction 2, which has an identical statement, has a transition with the same action a to to a state u' equivalent to s'.

(4) If a is the statement selecting *max*, then by definition of $\ominus$, the quantities $from[k].shoot[z][j] \ominus from[j].heal[z][k]$ are identical in states s and u; hence, set A is identical in these states. Similarly, the quantity $from[max].tag \odot from[j].tag$ is, by definition of $\odot$, identical in states s and u. Since Construction 2 has an identical statement, it follows that it has a transition with the same action a to a state u' equivalent to s'.

(5) If a is the conditional shoot increment, then the quantity $me.shoot[s][j] \ominus from[j].heal[s][i]$ is again identical in states s and u, and assigning $me.shoot[s][j] \oplus 1$ to $me.shoot[s][j]$ gives a state equivalent to the one Construction 2 reaches by assigning $me.shoot[s][j] + 1$ to the corresponding unbounded variable.

(6) Finally, if a is the selection of t, then because n divides $20n^2 + n$, we can define z uniquely as the number in $\{1, \ldots, n\}$ satisfying $z \equiv i - me.tag \pmod{n}$ in both state s and state u. Now in state s, t is chosen such that $t \odot me.tag = z$, and in state u it is chosen such that $t - me.tag = z$, so again the resulting states s' and u' are equivalent. □

5. *Complexity*

First let's consider the time complexity of Construction 3. Since all shared variable accesses occur in phases, each of which consists of n reads or writes in parallel, the parallel time complexity of both the Read and Write operation is bounded by the number of phases, which is clearly $O(1)$.

Next consider the space complexity of Construction 3, which is the size in bits of the type *shared*. This can be split into two parts: the data size and the control size.

The data size is $2 \times$ *sizeof* (ABS). This factor 2 overhead can be traced back to the use of single reader shared variables in our construction. In fact, the version of Construction 3 using multi-reader variables can be easily modified to do away with the *prev* data field (as sketched in the next section).

The control size concerns all the other fields in the shared variables. Note first that from the values read from $R_{j,i}$, user i never uses any of the counters $heal[1 - ss][k]$, where $k \neq i$. Thus, in addition to the single counter $heal[1 - ss][i]$, only a single heal counter set needs to be stored in $R_{j,i}$, of which the missing first index is understood to be $R_{j,i}.ss$. Thus, line h will change in each register $R_{i,j}$ only the $heal[1 - ss][j]$ field, and in line p, all heal counters in $R_{i,j}$ are changed together with $R_{i,j}.ss$. This leads to a control size of

$$\lceil \log(20n^2 + n) \rceil + 1 + (3n + 1)\lceil \log 14 \rceil \leq 12n + o(n).$$

6. *Subproblems*

Construction 3 presents a solution to the problem of implementing a multi-user variable from single-reader variables. Most other papers have considered the

intermediate level of a single-writer multi-reader variable, which splits the problem into two subproblems. We show that *projections* of our construction yield solutions to those two subproblems with competitive complexity measures.

The first projection is obtained by collapsing the row of shared variables $R_{i,0}, \ldots, R_{i,n-1}$ into a single multi-reader variable R_i. Each loop

$$\textbf{for } j \in I \textbf{ do } R_{i,j} := me$$

is replaced by the single write

$$R_i := me$$

while each read from $R_{j,i}$ is replaced by a read from R_j. The result is a solution to the problem of implementing a multi-user variable from single-writer multi-reader variables, since each of its histories corresponds to a history of Construction 3 in which all writes of a parallel loop happen to be consecutive events. The parallel time complexity is still constant, while the space complexity is $sizeof(shared) = 2 \times sizeof(\text{ABS}) + 16n + o(n)$. (Increase of $4n$ is due to the fact that now no heal counters can be omitted as was the case with the $R_{i,j}$'s.) The *prev* field can be made redundant by letting an aborting operation return $tmp[j].value$ instead of $tmp[j].prev$. Claim 3.3.7 is adjusted accordingly to the statement $a.s_j \prec w \preceq b \wedge tmp[j].value@a = value@b$, proven by choosing b equal to c instead of c^{-1}. The only modification needed to prove the Precedence part of Lemma 3.3.9 is that a' is chosen to be the Write from which $tmp[j].value@a$ originates, as this Write ends with a single .w event that necessarily precedes $a.t_j$. Otherwise, the proof of correctness remains unchanged.

The second projection is more involved, but yields a large space savings. Assume that only user 0 executes Write operations. Then all shoot counters of the remaining users, as well as all heal counters $heal[0 \ldots 1][1 \ldots n-1]$, remain 0, and can therefore be omitted. The *prev* field of the Read-only users will never be used and can also be omitted. Furthermore, 0 is easily seen to be always alive and therefore user 0 needs neither heal counters nor the shoot-selector *ss*. User 0 also never aborts and can always choose $max := 0$.

By Corollary 4.2.2, and because new tags are always chosen to be 0 (mod n), only tags $\{-10n, -9n, -8n, \ldots, 9n, 10n\}$ ever occur, and n can be factored out.

The result of removing all these redundancies is shown in Figure 4 as Construction 4.

The space complexity (taking into account possible savings) turns out to be $2 \times sizeof(\text{ABS}) + 8n + o(n)$ for the shared variables of user 0, and $sizeof(\text{ABS}) + O(1)$ for the shared variables of the remaining users. This space complexity is the same as that of Kirousis et al. [1987] and, apart from the data field used in the shared variables of Read only users, also the same as that of Peterson and Burns [1987], Newman-Wolfe [1987], and Singh et al. [1994].

7. *Conclusion*

Our construction shows that shared memory can be implemented, using replication, from simple bounded memory cells, with only a small constant factor increase in access time.

```
type I : 0..n − 1
type shared : record
                value,prev : ABStype
                tag  : −10..10
                ss : 0..1
                shoot : array[0..1][0..n − 1] of −4..9
                heal : array[0..1][0] of −4..9
              end
```

```
procedure Write(v)
var j : I
    from : array[0..n − 1] of shared
    static me : shared
begin
    for j ∈ I do from[j] := R_{j,0}
    me.prev := me.value
    for j ∈ I do R_{0,j} := me
    for j ∈ I, s ∈ {0..1} do
        if me.shoot[s][j] ⊖ from[j].heal[s][0] < 6 then me.shoot[s][j]⊕ := 1
    me.value, me.tag := v, (me.tag + 11) mod 21 − 10
    for j ∈ I do R_{0,j} := me
end
```

```
procedure Read(i) (i = 1..n − 1)
var j : I
    s : 0..1
    from,tmp : array[0..n − 1] of shared
    static me : shared
begin
    s := 1 − me.ss
    me.heal[s][0] := R_{0,i}.shoot[s][i]
    R_{i,0} := me
    for j ∈ I do from[j] := R_{j,i}
    tmp[0] := R_{0,i}
    if tmp[0].shoot[s][i] ⊖ me.heal[s][0] ≥ 3 then return tmp[0].prev
    select max ∈ A such that ∀j ∈ A : from[max].tag ⊙ from[j].tag ≥ 0
    where A = {0} ∪ {j ∈ I : from[0].shoot[z][j] ⊖ from[j].heal[z][0] < 6
                                where z = from[j].ss}
    me.value, me.tag, me.ss := from[max].value, from[max].tag, s
    for j ∈ I do R_{i,j} := me
    return me.value
end
```

FIG. 4. Construction 4; Single- to multi-reader.

To this end, the unbounded solution of Vitányi and Awerbuch [1986], for a long time the only recognized correct solution, is refined by adding a powerful, in itself unbounded, shooting mechanism. This mechanism allows slow, potentially confused operations to safely abort, and allows the remaining operations to interpret the unbounded timestamps of Vitányi and Awerbuch as bounded quantities. The end result follows by showing that the shooting mechanism itself is easily bounded.

ACKNOWLEDGMENTS. We thank Amos Israeli and Leslie Lamport for fruitful discussions. The numerous useful suggestions by the anonymous referees greatly helped to improve the presentation.

REFERENCES

ANDERSON, J. 1993. Composite registers. *Dist. Comput. 6*, 1993, 141–154.

AWERBUCH, B., KIROUSIS, L., KRANAKIS, E., AND VITÁNYI, P. M. B. 1988. A proof technique for register atomicity. In *Proceedings of the 8th Conference on Foundations of Software Technology and Theoretical Computer Science*. Lecture Notes in Computer Science, Vol. 338. Springer Verlag, Heidelberg, Germany, pp. 286–303.

BLOOM, B. 1988. Constructing two-writer atomic registers. *IEEE Trans. Comput. 37*, 1506–1514.

BURNS, J. E., AND PETERSON, G. L. 1987. Constructing multi-reader atomic values from nonatomic values. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing* (Vancouver, B. C., Canada, Aug. 10–12). ACM, New York, pp. 222–231.

DOLEV, D., AND SHAVIT, N. 1996. Bounded concurrent time-stamp systems are constructible. *SIAM J. Comput.*, to appear.

DWORK, C., HERLIHY, M., PLOTKIN, S., AND WAARTS, O. 1992. Time-lapse snapshots. In *Proceedings of the 1st Israel Symposium on Theory of Computing and Systems*. pp. 154–170.

DWORK, C., AND WAARTS, O. 1992. Simple and efficient bounded concurrent timestamping, or, Bounded concurrent timestamp systems are comprehensible! In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing* (Victoria, B.C., Canada, May 4–6). ACM, New York, pp. 655–666.

GAWLICK, R., LYNCH, N., AND SHAVIT, W. 1992. Concurrent timestamping made simple. In *Proceedings of the 1st Israel Symposium on Theory of Computing and Systems*. pp. 171–183.

HALDAR, S., AND VIDYASANKAR, K., 1992. Counterexamples to a one writer multireader atomic shared variable construction of Burns and Peterson. *ACM Oper. Syst. Rev. 26*, 1, 87–88.

HALDAR, S., AND VIDYASANKAR, K. 1995. Constructing 1-writer multireader multivalued atomic variables from regular variables. *J. ACM 42*, 1 (Jan.), 186–203.

HERLIHY, M. P. 1988. Impossibility and universality results for wait-free synchronization. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing* (Toronto, Ont., Canada, Aug. 15–17). ACM, New York, pp. 276–290.

HERLIHY, M. P., AND WING, J. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst. 12*, 3 (July), 463–492.

ISRAELI, A., AND LI, M. 1987. Bounded time-stamps. In *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*. IEEE, New York, pp. 371–382.

ISRAELI, A., AND PINHASOV, M. 1992. A concurrent time-stamp scheme which is linear in time and space. In *Proceedings of the 6th International Workshop on Distributed Algorithms*. Lecture Notes in Computer Science, vol. 647. Springer-Verlag, Heidelberg, Germany, pp. 95–109.

ISRAELI, A., AND SHAHAM, A. 1992. Optimal multi-writer multi-reader atomic register. In *Proceedings of the 11th annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B. C., Canada, Aug. 10–12). ACM, New York, pp. 71–82.

KIROUSIS, L. M., KRANAKIS, E., AND VITÁNYI, P. M. B. 1987. Atomic multireader register. In *Proceedings of the 2nd International Workshop on Distributed Computing*. Lecture Notes in Computer Science, vol. 312. Springer-Verlag, New York, pp. 278–296.

LAMPORT, L., 1986. On Interprocess Communication Parts I and II. *Dist. Comput. 1*, 77–101.

LI, M., AND VITÁNYI, P. M. B. 1988. A very simple construction for atomic multiwriter register. Tech. Rep. TR 01-87. Aiken Comp. Lab., Harvard Univ. Nov.

LI, M., AND VITÁNYI, P. M. B. 1989. How to share concurrent asynchronous wait-free variables. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*. Lecture Notes in Computer Science, vol. 372. Springer-Verlag, New York, pp. 488–505.

LI, M., AND VITÁNYI, P. M. B. 1992. Optimality of wait-free atomic multiwriter variables. *Inf. Proc. Lett. 43*, 107–112.

LI, M., TROMP, J., AND VITÁNYI, P. M. B. 1989. How to share concurrent wait-free variables, Tech. Rep. CS-8916. CWI, Amsterdam, The Netherlands, April.

LYNCH, N. A., AND VAANDRAGER, F. W. 1995. Forward and backward simulations. Part I: Untimed systems, *Inf. Comput. 121*, 2, 214–233.

NEWMAN-WOLFE, R. 1989. A protocol for wait-free, atomic, multi-reader shared variables. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B. C., Canada, Aug. 10–12). ACM, New York, pp. 232–248.

PETERSON, G. L. 1983. Concurrent reading while writing. *ACM Trans. Prog. Lang. Syst. 5*, 1 (Jan.), 46–55.

PETERSON, G. L., AND BURNS, J. E. 1987. Concurrent reading while writing II: The multiwriter case. In *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*. IEEE, New York, pp. 383–392.

SCHAFFER, R. 1988. On the correctness of atomic multi-writer registers, Tech. Rep. MIT/LCS/TM-364. MIT Lab. for Computer Science, MIT, Cambridge, Mass., June.

SINGH, A. K., ANDERSON, J. H., AND GOUDA, M. G. 1994. The Elusive Atomic Register. *J. ACM 41*, 2 (Mar.), 311–339.

TROMP, J. 1989. How to Construct an Atomic Variable. In *Proceedings of the 3rd International Workshop on Distributed Algorithms*. Lecture Notes in Computer Science, vol. 392. Springer-Verlag, Heidelberg, Germany, pp. 292–302.

VIDYASANKAR, K. 1988. Converting Lamport's Regular Register to an atomic register. *Inf. Proc. Lett. 28*, 287–290.

VITÁNYI, P. M. B., AND AWERBUCH, B. 1986. Atomic shared register access by asynchronous hardware. In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*. IEEE, New York, 1986, pp. 233–243. (Errata, Ibid.,1987)

RECEIVED APRIL 1989; REVISED NOVEMBER 1995; ACCEPTED FEBRUARY 1996