Simple Wait-free Multireader Registers
(Extended Abstract)

Paul Vitányi[1]

Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands. Email: paulv@cwi.nl. This work was supported in part by the EU fifth framework project QAIP, IST–1999–11234, the NoE QUIPROCONE IST–1999–29064, the ESF QiT Programmme, and the EU Fourth Framework BRA NeuroCOLT II Working Group EP 27150.

Abstract. Multireader shared registers are basic objects used as communication medium in asynchronous concurrent computation. We propose a surprisingly simple and natural scheme to obtain several wait-free constructions of bounded 1-writer multireader registers from atomic 1-writer 1-reader registers, that is easier to prove correct than any previous construction. Our main construction is the first symmetric pure timestamp one that is optimal with respect to the worst-case local use of control bits; the other one is optimal with respect to global use of control bits; both are optimal in time.

1 Introduction

Interprocess communication in distributed systems happens by either message-passing or shared variables (also known as shared *registers*). Lamport [19] argues that every message-passing system must hide an underlying shared variable. This makes the latter an indispensable ingredient. The multireader wait-free shared variable is (even more so than the multiwriter construction) the building block of virtually all bounded wait-free shared-variable concurrent object constructions, for example, [28, 15, 5, 8, 6, 7, 13, 2, 16, 24]. Hence, understanding, simplicity, and optimality of bounded wait-free atomic multireader constructions is a basic concern. Our constructions are really simple and structured extensions of the basic unbounded timestamp algorithm in [28], based on a natural recycling scheme of obsolete timestamps.

Asynchronous communication: Consider a system of asynchronous processes that communicate among themselves by executing read and write operations on a set of shared variables only. The system has no global clock or other synchronization primitives. Every shared variable is associated with a process (called *owner*) which writes it and the other processes may read it. An execution of a write (read) operation on a shared variable will be referred to as a *Write* (*Read*) on that variable. A Write on a shared variable puts a value from a pre determined finite domain into the variable, and a Read reports a value from the domain. A process that writes (reads) a variable is called a *writer* (*reader*) of the variable.

Wait-free shared variable: We want to construct shared variables in which the following two properties hold. (1) Operation executions are not necessarily atomic, that is, they are not indivisible, and (2) every operation finishes its execution within a bounded number of its own steps, irrespective of the presence of other operation executions and their relative speeds. That is, operation executions are *wait-free*. These two properties give rise to a classification of shared variables: (i) A *safe* variable is one in which a Read not overlapping any Write returns the most recently written value. A Read that overlaps a Write may return any value from the domain of the variable; (ii) A *regular* variable is a safe variable in which a Read that overlaps one or more Writes returns either the value of the most recent Write preceding the Read or of one of the overlapping Writes; and (iii) An *atomic* variable is a regular variable in which the Reads and Writes behave as if they occur in some total order which is an extension of the precedence relation.

Multireader shared variable: A multireader shared variable is one that can be written by one process and read (concurrently) by many processes. Lamport [19] constructed an atomic wait-free shared variable that could be written by one process and read by one other process, but he did not consider constructions of wait-free shared variables with more than one writer or reader.

Previous Work: In 1987 there appeared at least five purported solutions for the wait-free implementation of 1-writer n-reader atomic shared variable from atomic 1-writer 1-reader shared variables: [18, 27, 4, 26] and the conference version of [15], of which [4] was shown to be incorrect in [9] and only [26] appeared in journal version. The only other 1-writer n-reader atomic shared variable constructions appearing in journal version are [10] and the "projection" of the main construction in [21]. A selection of related work is [28, 21, 3, 4, 10–12, 17–21, 23–27]. A brief history of atomic wait-free shared variable constructions and timestamp systems is given in [13].

Israeli and Li [15] introduced and analyzed the notion of *timestamp system* as an abstraction of a higher typed communication medium than shared variables. (Other constructions are [5, 16, 8, 6, 7, 13].) As an example of its application, [15] presented a non-optimal multireader construction, partially based on [28], using order n^3 control bits overall, and order n control bits locally for each of $O(n^2)$ shared 1-reader 1-writer variables. Our constructions below are inspired by this timestamp method and exploit the limited nature of the multireader problem to obtain both simplification and optimality.

This work: We give a surprisingly simple and elegant wait-free construction of an atomic multireader shared variable from atomic 1-reader 1-writer shared variables. Other constructions (and our second construction) don't use pure timestamps [15] but divide the timestamp information between the writer's control bits and the reader's control bits. Those algorithms have an asymmetric burden on the writer-owned subvariables. Our construction shows that using pure timestamps we can balance the size of the subvariables evenly among all owners, obtaining a symmetric construction. The intuition behind our approach is given in Section 2. Technically:

(i) Our construction is a bounded version of the unbounded construction in [28] restricted to the multireader case (but different from the "projection" of the bounded version in [21]), and its correctness proof follows simply and directly from the correctness of the unbounded version and a proof of proper recycling of obsolete timestamps.

(ii) The main version of our construction is the first construction that uses a sublinear number of control bits, $O(\log n)$, locally *for everyone* of the n^2 constituent 1-reader 1-writer subvariables. It is easy to see that this is optimal locally, leading to a slightly non-optimal global number of control bits of order $n^2 \log n$. Implementations of wait-free shared variables assume atomic shared (control) subvariables. Technically it may be much simpler to manufacture in hardware—or implement in software—small $\Theta(\log n)$-bit atomic subvariables that are robust and reliable, than the exponentially larger $\Theta(n)$-bit atomic subvariables required in all previous constructions. A wait-free implementation of a shared variable is fault-tolerant against crash failures of the subvariables, but not against more malicious, say Byzantine, errors. Slight suboptimality is a small price to pay for ultimate reliability.

(iii) Another version of our construction uses n 1-writer 1-reader variables of n control bits and n^2 1-writer 1-reader variables of 2 control bits each, yielding a global $O(n^2)$ number of control bits. There need to be n^2 1-reader 1-writer subvariables for pairwise communication, each of them using some control bits. Hence the global number of control bits in any construction is $\Omega(n^2)$. We also reduce the number of copies of the value written to $O(n)$ rather than $O(n^2)$. All these variations of our construction use the minimum number $O(n)$ of accesses to the shared 1-reader 1-writer subvariables, per Read and Write operation.

Construction	Global control bits	Max local control bits
[18]	$\Theta(n^3)$	$\Theta(n)$
[27]	$\Theta(n^2)$	$\Theta(n)$
[4]-incorrect: [9]	$\Theta(n^2)$	$\Theta(n)$
[26]	$\Theta(n^2)$	$\Theta(n)$
[15]	$\Theta(n^3)$	$\Theta(n)$
[21]	$\Theta(n^2)$	$\Theta(n)$
[10]	$\Theta(n^2)$	$\Theta(n)$
This paper (ii)	$\Theta(n^2 \log n)$	$\Theta(\log n)$
This paper (iii)	$\Theta(n^2)$	$\Theta(n)$

Table 1. Comparison of Results.

1.1 Model, Problem Definition, and some Notations

Throughout the paper, the n readers and the single writer are indexed with the set $I = \{0, \ldots, n\}$—the writer has index n. The multireader variable constructed will be called ABS (for abstract).

A construction consists of a collection of atomic 1-reader 1-writer 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*, to execute a read opeartion or write operation on ABS respectively. 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. A return statement must end both procedures, in the case of Read having an argument which 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 inter-process 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 lower case. 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.

1.2 Correctness

A wait-free 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. 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 readers of the constructed variable ABS. A state has precisely $n+1$ immediate successors: there is at precisely one atomic statement per process to be executed next (each process is deterministic).

A *history* of the construction is a finite or infinite sequence of states $t_0, t_1, \ldots$ 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.

An event a *precedes* an event b in history h, $a \prec_h b$, if a occurs before b in h. 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 $a \preceq_h b$ to denote that either $a =_h b$ or $a \prec_h b$. The relation $\prec_h$ on event-sets constitutes what is known as an *interval order*. That is, a partial order $\prec$ 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 [19] 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 final **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 [14]. 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. We use the following definition and lemma from [21]:

Definition 1. *A construction is* correct *if it satisfies Wait-Freedom and all its complete proper histories are linearizable.*

Lemma 1. *A complete proper history h is linearizable iff there exists a function mapping each operation in h to a rational number, called its timestamp, such that the following 3 conditions are satisfied:*

Uniqueness: *different Write operations have different timestamps.*

Integrity: *for each Read operation there exists a Write operation with the same timestamp and value, that it doesn't precede.*

Precedence: *if one operation precedes another, then the timestamp of the latter is at least that of the former.*

2 Intuition

We use the following intuition based on timestamp systems: The concurrent reading and writing of the shared variable by one writer and n readers (collectively, the *players*) is viewed as a pebble game on a finite directed graph. The nodes of the graph can be viewed as the timestamps used by the system, and a pebble on a node as a subvariable containing this timestamp. If the graph contains an arc pointing from node a to node b then a is *dominated* by b ($a < b$). The graph has no cycles of length 1 or 2. First, suppose that every player has a

single pebble which initially is placed at a distinguished node that is dominated by all other nodes. A Read or Write by a player consists in observing where the pebbles of the other players are, and determining a node to move its own pebble to. In a Write, the writer puts its pebble on a node that satisfies all of the following: (i) it dominates the previous position; (ii) it is not occupied by a pebble of a reader; and (iii) it is not dominated by a node occupied by a pebble of a reader. In a Read, the reader concerned puts its pebble at a node containing the writer's pebble. In the pebble game a player can only observe the pebble positions of the other players in sequence, one at a time. By the time the observation sequence is finished, pebbles observed in the beginning may have moved again. Thus, in an observation sequence by a reader, a node can contain another reader's pebble that dominates the node containing the writer's pebble. Then, the second reader has already observed the node pebbled by a later Write of the writer and moved its pebble there. In the unbounded timestamp solution below, where the timestamps are simply the nonnegative integers and a higher integer dominates a lower integer, this presents no problem: a reader simply moves its pebble to the observed pebbled node corresponding to the greatest timestamp. But in the bounded timestamp solution we must distinguish obsolete timestamps from active recycled timestamps. Clearly, the above scenario cannot happen if in a Read we observe the position of the writer's pebble last in the obervation sequence. Then, the linearization of the complete proper history of the system consists of ordering every Read after the Write it joined its pebble with. This is the basic idea. But there is a complication that makes life less easy.

In the implementation of the 1-writer n-reader variable in 1-writer 1-reader subvariables, every subvariable is a one-way communication medium between two processes. Since two-way communication between every pair of processes is required, there are at least $(n+1)n$ such subvariables: Every process owns at least n subvariables it can write, each of which is read by one of the other n processes. Thus, a pebble move of a player actually consists in moving (at least) n pebble copies, in sequence one at a time—a move of a pebble consists in atomically writing a subvariable. Every pebble copy can only be observed by a single fixed other player—the reader of that 1-writer 1-reader subvariable. This way player 0 may move the pebble copy associated with player 1 to another node (being observed by player 1 at time t_1) while the pebble copy associated with player 2 is still at the originating node (being observed later by player 2 at time $t_2 > t_1$). This once more opens the possibility that a reader sees a writer's pebble at a node pebbled by a Write, while another reader earlier on sees a writer's pebble at a node pebbled by a later Write. The following argument shows that this 'later Write' must be in fact the 'next Write': A reader always joins its pebble to a node that is already pebbled, and only the writer can move its pebbles to an unoccupied node. Therefore, the fact that a reader observes the writer's pebble last guaranties that no reader, of which the pebble was observed before, can have observed a Write's pebble position later than the next Write. Namely, at the time the writer's pebble was observed by the former reader at the node pebbled by Write, the next Write wasn't finished, but the latter reader has already observed

the writer's pebble. This fact is important in the bounded timestamp solution we present below, since it allows us to use a very small timestamp graph that contains cycles of length just 3: $G = (V, E)$ with $V = \{\perp, 1, \ldots, 4n-3\}^2$ and $(v_1, v_2) \in E$ $(v_1 < v_2)$ iff either $v_1 = (i, j), v_2 = (k, h)$ and $j = k$ and $i \neq h \neq \perp$, or $(i, j) = (\perp, \perp)$ and $h \neq \perp$.

This approach suffices to linearize the complete proper history of the system if we can prevent a Write to pebble the same node as a Read in the process of chasing an older obsolete Write. This we accomplish by a Read by player i announcing its intended destination node several times to the writer, with an auxiliary pebble for the purpose (special auxiliary subvariable) intended for this purpose, and in between checking whether the writer has moved to the same node. In this process either the Read discovers a Write that is intermediate between two Writes it observed, and hence covered by the Read, in which case it can safely report that Write and choose the bottom timestamp $(\perp, \perp)$. Alternatively, the Read ascertains that future Writes know about the timestamp it intends to use and those Writes will avoid that timestamp.

3 Unbounded

Figure 1 shows Construction 0, which is a restriction to the multireader case of the unbounded solution multiwriter construction of [28]. Line 2 of the Write procedure has the same effect as "*free* := *free* + 1" with *free* initialized at 0 (because the writer always writes *free* to $R_{n,n}.tag$ in line 4). The processes indexed $0, \ldots, n-1$ are the readers and the process indexed n is the writer. We present it here as an aid in understanding Construction 1. Detailed proofs of correctness (of the unrestricted version, where every process can write and not just process n) are given in [28] and essentially simple proofs in [21] and the textbook [22].

The timestamp function called for in lemma 1 is built right into this construction. Each operation starts by collecting value-timestamp pairs from all users. In line 3 of either procedure, the operation picks a value and timestamp for itself. It finishes after distributing this pair to all users. It is not hard to see that the three conditions of lemma 1 are satisfied for each complete proper history. Integrity and Precedence are straightforward to check. Uniqueness follows since timestamps of Write operations of the single writer strictly increase (based on the observation that each $R_{i,i}.tag$ is nondecreasing).

Restricting our unbounded timestamp multiwriter algorithm of [28], in the version of [21], to the single writer case enabled us to tweak it to have a new very useful property that is unique to the multireader case: The greatest timestamp scanned by a reader is either the writer's timestamp *from*$[n].tag$ or another reader's timestamp that is at most 1 larger. The tweaking consists in the fact that there is a definite order in the scanning of the shared variables in line 2 of the Read procedure: the writer's shared variable is scanned last (compare Section 2).

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

```
procedure Write(n, v)
var j : I
    free : integer
    from : array[0..n] of shared
begin
1   for j := 0..n do from[j] := R_{j,n}
2   free := max_{j∈I} from[j].tag + 1
3   from[n] := (v, free)
4   for j := 0..n do R_{n,j} := from[n]
end
```

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

Fig. 1. Construction 0

Lemma 2. *The* max *selected in line 2 of the Read procedure satisfies* $from[n].tag \leq from[max].tag \leq from[n].tag + 1$.

Proof. At the time the writer's shared variable $R_{n,i}$ is scanned last in line 2 of the $Read(i)$ procedure, yielding $from[n].tag$, the writer can have started its next write, the $(from[n].tag+1)$th Write, but no write after that—otherwise the $from[n].tag$ would already have been overwritten in $R_{n,i}$ by the $(from[n].tag + 1)$th Write. Hence, a timestamp scanned from another reader's shared variables $R_{j,i}$ $(j \neq n, i)$ can exceed the writer's timestamp by at most 1. □

4 Bounded

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

Our strategy will be as follows. We will stick very close to construction-0 and only modify or expand certain lines of code. The new bounded timestamps will consist of two fields, like dominoes, the *tail* field and *head* field.

Definition 2. *A* bounded timestamp *is a pair (p, c) where p is the value of the* tail *field and c is the value of the* head *field. Every field can contain a value t with $0 \leq t \leq 4n+2$ or t is the distinguished initial, or bottom, value $\perp$. ($\perp$ is not a number and is lower than any number.) We define the domination relation "$<$" on the bounded timestamps as follows: $(p_1, c_1) < (p_0, c_0)$ if either $c_1 = p_0$ and $p_1 \neq c_0 \neq \perp$, or $p_1, c_1 = \perp$ and $c_0 \neq \perp$.*

The matrix of shared variables stays the same but for added shared variables $R_{i,n+1}$ for communication between readers i $(0 \leq i \leq n-1)$ and the writer n. These shared variables are used by the readers to inform the writer of all timestamp values that are not obsolete, and hence cannot be recycled yet. The interesting part of the later proof is to show that this statement is true. The scan executed in line 1 of the Write protocol gathers all timestamps written in the $R_{i,n}$'s $(0 \leq i \leq n)$ and $R_{i,n+1}$'s $(0 \leq i \leq n-1)$, the timestamps that are not obsolete, and hence the process can determine the $\leq 4n+2$ values occurring in the fields (two per timestamp), and select a value that doesn't occur (there are $4n+3$ values available exclusive of the bottom value $\perp$). The initial state of the construction has all $R_{i,j}$ containing $(0, \perp, \perp)$.

The lines of Construction-1 are numbered maintaining—or subnumbering—the corresponding line numbers of Construction-0. The only difference in the Write protocols is line 2 (select new timestamp). In the Read protocols the differences are lines 2.x (determine latest—or appropriate—timestamp) and lines 3.x (assign selected timestamp to local variable *from*$[i]$), and lines 0.x (start Read by reading writer's timestamp and writing it back to writer). According to this scheme we obtain the out-of-order line-sequence 2.4, 3.1, 2.5, 3.2 because the instructions in lines 2 and 3 of Construction-0 are split and interleaved in Construction-1.

Lemma 3. *Line 2 of a Write always selects a free integer c_1 such that the timestamp (p_1, c_1) with new head c_1 and new tail p_1 (head of the timestamp of the directly preceding Write), assigned in line 3 and written in line 4 of that Write, satisfies $(p_1, c_1) \not< (p_0, c_0)$ and $(p_1, c_1) \neq (p_0, c_0)$ for every timestamp (p_0, c_0) either scanned in line 1 of the Write, or presently occurring in a local variable of a concurrent Read that will eventually be written to a shared variable in line 4 of that Read.*

Proof. In line 1 of a Write the shared variables with each reader, and a redundant shared variable with itself, are scanned in turn. In assigning a value to *free* in line 2 the writer avoids all values that occur in the *head* and *tail* fields of the shared variables. All local variables of a reader are re-assigned from shared variables in executing the Read procedure, before they are written to shared variables. So the only problem can be that *free* occurs in a local variable of a concurrently active Read that has not yet written it to a variable shared with the writer

```
    shared : record
                value : ABStype
                tail  : ⊥, 0..4n + 2
                head  : ⊥, 0..4n + 2
             end
```

```
procedure Write(n, v)
var j : 0..n
    free : 0..4n + 2
    from : array[0..n] of shared
    notfree : array[0..n − 1] of shared
begin
1   for j := 0..n do from[j] := R_{j,n}; for j := 0..n − 1 do notfree[j] := R_{j,n+1}
2   free := least positive integer not in tail or head fields of from or notfree records
3   from[n] := (v, from[n].head, free)
4   for j := 0..n do R_{n,j} := from[n]
end
```

```
procedure Read(i)
var j, max
    temp : shared
    from : array[0..n] of shared
begin
0.1 temp := R_{n,i}
0.2 R_{i,n+1} := temp
1   for j = 0..n do from[j] := R_{j,i}
2.1 if from[n] ≠ temp then
2.2         temp, R_{i,n+1} := from[n]
2.3         for j = 0..n do from[j] := R_{j,i}
2.4         if from[n] ≠ temp then
3.1           from[i] := (temp.value, ⊥, ⊥); goto 4
2.5 if ∃max : from[max].(tail, head) > from[n].(tail, head) then
3.2         from[i] := from[max]
3.3 else from[i] := from[n]
4   for j := 0..n do R_{i,j} := from[i]
5   return from[i].value
end
```

Fig. 2. Construction 1

at the time that variable was scanned by the Write. If *free* exists in a local variable *temp* this doesn't matter since the timestamp concerned will not be used as a Read timestamp. Hence, the only way in which *free* can be assigned a value that concurrently exists in a local variable of a Read which already has been used or eventually can be used in selection line 2.5 of a Read, and hence eventually can be part of an offending timestamp written to a shared variable, is according to the following scenario: A $Read(i)$, say R, is active at the time a Write, say W_1, reads either one of their shared variables. This R will select or has already selected the offending timestamp (p_0, c_0), originally written by a Write W_0 preceding W_1, but R has not yet written it to $R_{i,n+1}$ or $R_{i,n}$ by the times W_1 scans those variables. Moreover, W_1 will in fact select a timestamp (p_1, c_1) with either $(p_1, c_1) < (p_0, c_0)$ or $(p_1, c_1) = (p_0, c_0)$. This can only happen if *free* in line 2 is p_0, or c_0 and the $p_0 = head$ of the timestamp of the Write immediately preceding W_1, respectively. Then, a later Read R_0 using in its line 2.5 the (p_1, c_1) timestamp written by W_1 and the (p_0, c_0) timestamp written by R concludes falsely that W_1 precedes W_0 or $W_1 = W_0$. For this to happen, R has to write (p_0, c_0) in line 4, and has to assign it previously in one of lines 3.1, 3.2, or 3.3. In line 3.1 the timestamp assigned is the bottom timestamp $(\perp, \perp)$ which by definition cannot dominate or equal a timestamp assigned by the writer. This leaves assignement of the offending timestamp (p_0, c_0) in line 3.2 or line 3.3. Without loss of generality, assume that the domination and equality of the timestamps used by R in line 2.5 is still correct (so R is a "first" Read writing an offending timestamp).

Case 1: Timestamp (p_0, c_0) is assigned in line 3.2 of R and hence won the competition in line 2.5 by $from[n].(tail, head) < (p_0, c_0)$. Let W be the Write that wrote $from[n].(tail, head)$. We first show that W directly precedes W_0. Since $from[n].(tail, head) < (p_0, c_0)$ we have $W \neq W_0$. If W follows W_0, then the conclusion from $from[n].(tail, head) < (p_0, c_0)$ that W_0 is the next Write after W is false (and (p_0, c_0) has already been written by a $Read(j)$, $j \neq i$, to the shared variable $R_{j,i}$ from which R scanned it). This contradicts the assumption above that domination and equality of timestamps used by R in line 2.5 is still correct. The only remaining possibility is that W precedes W_0. Of the timestamps used in the competition in line 2.5, the timestamp written by W was read last. Therefore, with W preceding W_0, W directly precedes W_0 and W wrote a timestamp $(\cdot, p_0)$, as is in fact evidenced by the relation $from[n].(tail, head) < (p_0, c_0)$.

Case 1.a: R executes line 2.5, and the test in line 2.1 was 'true'—a Write wrote shared variable $R_{n,i}$ in between line 0.1 and line 2.1—and the test in line 2.4 was 'false'—no Write wrote $R_{n,i}$ in between line 1 and line 2.4—and $from[n].(tail, head)$ used in line 2.5 is the one read in line 1 and written by Write W in between line 0.1 and line 2.1. Therefore, Write W_0 started after line 0.1 but before line 2.4 (since (p_0, c_0) came by way of a $Read(j)$ with $j \neq i$ in line 2.3), and W_0 had not yet written $R_{n,i}$ before line 2.4. Then, every Write following W_0 started after line 2.3. As a consequence, such a Write both reads the timestamp written to $R_{i,n+1}$ in line 2.2 of R, and avoids choosing $free := p_0$ $(= from[n].head$ originating from $W)$ in its own line 2 (until $R_{i,n+1}$ is overwritten again). Hence

the *tail* value in (p_0, c_0) is not re-used and the timestamp is actually not offensive. (Since $c_0 \neq p_0$ by the selection of *free* in line 2 of the Write protocol, all Writes following W_0 cannot create a timestamp $(\cdot, p_0)$ and hence also not (p_0, c_0).)

Case 1.b: R executes line 2.5, and the test in line 2.1 was 'false'—no Write wrote shared variable $R_{n,i}$ in between line 0.1 and line 2.1—and $from[n].(tail, head)$ used in line 2.5 is the one read in line 1 and written by Write W in between line 0.1 and line 2.1. The remainder of the argument is the same as in Case 1.a.

Case 2: Timestamp (p_0, c_0) is assigned in line 3.3 of R.

Case 2.a: Timestamp (p_0, c_0) was already scanned in line 0.1 of R, and again in line 1, and written by a Write W_0 writing the shared variable $R_{n,i}$ before the scan of line 0.1. If W_1 is the next Write after W_0, then every Write following W_1 will read (p_0, c_0) and avoid both its values in selecting *free* in line 2 and assigning the *head* of its new timestamp in line 3, because (p_0, c_0) was written in line 0.2 to $R_{i,n+1}$ before W_1 finished (and hence before a later Write started)—otherwise the timestamp of W_1 would have been observed in the scan of line 1. But W_1 will do this as well, since (p_0, c_0) is the timestamp written by W_0, and hence observed by W_1. Consequently, no value in (p_0, c_0) is re-used as the *head* of a new timestamp by a Write (until $R_{i,n+1}$ is overwritten again and the timestamp disappears) and (p_0, c_0) is actually not offensive.

Case 2.b: Timestamp (p_0, c_0) was first scanned in line 1 of $Read(i)$—the contents of $R_{n,i}$ scanned in line 0.1 is different from that scanned in line 1. Therefore, (p_0, c_0) was written by Write W_0 writing the shared variable $R_{n,i}$ after the scan at line 0.1. The timestamp scanned in line 2.3 was still the one written by W_0 (otherwise the test in line 2.4 would be positive and result in assignment 3.1 that doesn't assign the offending timestamp at all). The remainder of this case is identical with last part of Case 2.a starting from "If W_1 is the next Write after W_0". □

The crucial feature that makes the bounded algorithm work is the equivalent of lemma 2:

Lemma 4. *The timestamp selected in lines 2.x and written in line 4 of a* $Read(i)$ *is one of the following:*

(i) The timestamp $(\perp, \perp)$ *for a Write that is completely overlapped by the Read concerned;*

(ii) Another reader's timestamp scanned in line 1 (respectively, line 2.3) that is written by the next Write after the Write that wrote the timestamp $from[n].(tail, head)$ *scanned in line 1.*

(iii) Otherwise, the writer's timestamp $from[n].(tail, head)$ *scanned in line 1.*

Proof. Only assignments in lines 3.1, 3.2, 3.3 can result in a write in line 4.

(i) If line 3.1 is executed in a Read, then previously we scanned the writer's timestamp in lines 0.1, 1, 2.3, and obtained three successive timestamps without two successive ones being the same. Hence the Write corresponding to the middle scan, of line 1, is overlapped completely by the Read, and the Read can be ordered directly after this Write, and this has no consequences for the remaining

ordering. This is reflected by using the timestamp $(\perp, \perp)$ to be written by such a Read in line 4.

(ii) If line 3.2 is executed in a Read then the timestamp assigned is a reader's timestamp scanned in line 1 or line 2.3. The writer's timestamp used in the comparison line 2.5 is, say, (t, h). According to the semantics of the "$<$" sign in definition 2, only a reader's timestamp of the form (h, free) satisfies the condition in line 2.5. The only timestamps in the system are created by the writer. By Lemma 3, existence of the (h, free) timestamp somewhere in the system at the time of writing the current instance of timestamp (t, h) would have prevented the writer from writing (t, h). Thus the writer must have written the (h, free) timestamp after it wrote (t, h). Using Lemma 3 a second time, the writer can write a timestamp with h in the *tail* field only at the very next Write after the Write that wrote a timestamp with h in the *head* field, since the (t, h) timestamp is still somewhere in a shared variable or to be written to a shared variable.

(iii) If line 3.3 is executed in a Read, then items (i) and (ii) were not applicable and the timestamp assigned is the writer's timestamp scanned in line 1. □

Theorem 1. *Construction-1 is a wait-free implementation of an atomic 1-writer n-reader register. It uses $(n+1)(n+2)-1$ atomic 1-writer 1-reader $2\log(4n+4)$ bits control shared variables. The Write scans $2n+1$ of these variables and writes $n+1$ of them; the Read scans $\leq 2n+3$ of these variables and writes $\leq n+3$ of them.*

Proof. **Complexity:** Since the program of Construction-1 contains no loops, it is straightforward to verify that every Read and Write executes the number of accesses to shared variables, and the size of the shared variables, as in the statement of the theorem.

Wait-Freedom: This follows directly from the upper bounds on the complexity (scans and writes) of the shared variables in the construction, and the fact that the program of Construction-1 is loop-free.

Linearizability: Consider a complete proper history (as defined before) h of all Writes and only those Read's that don't write the bottom timestamp $(\perp, \perp)$ in line 4. Let $\prec_h$ be the partial order induced by the timing of the Reads and Writes of h. Lemma 4, items (ii) and (iii), asserts that on this history h, Construction-0 and Construction-1 behave identically in that the same Read reports the values of the same Write in both constructions. Therefore, with respect to h, linearizability of Construction-1 follows from the linearizability of Construction-0. Let $\prec_h^l$ be the linear order thus resulting from $\prec_h$. The remaining Read's, those that write the bottom timestamp $(\perp, \perp)$ in line 4, completely overlap a Write, lemma 4 item (i), and report the value of that overlapped Write. Hence they can, without violating linearizability, be inserted in the $\prec_h^l$-order directly following the Write concerned. This shows that Construction-1 is linearizable. □

Minimum Number of Global Control Bits: The same algorithm with only $O(n^2)$ control bits overall can be constructed as follows. Each register owned by the writer contains $2n$ control bits, and each register owned by a reader contains only 2 control bits. The control bits are used to determine the domination

relation between readers and the writer. The Protocol stays the same, only the decisions in the protocol are made according to different format data. Since the decisions are isomorphic with that of Protocol 1, the correctness of the new Protocol follows by induction on the total atomic order of the operation executions in each run by the correctness of Construction-1.

Minimum Number of Replicas of Stored Values: In the algorithm, each subregister ostensibly contains a copy of the value to be written. This sums up to $O(n^2 \log V)$ bits, for the value ranging from 1 to V. With the following scheme only the registers owned by the writer contain the values. Each register owned by the writer can contain two values. The two fields concerned are used alternatingly. The writer starts its tth write with an extra write (line 0) to all registers it owns, writing the new value in field $t \pmod 2$. In line 4 it writes to field $t \pmod 2$ (it marks this field as the last one written), and finishes by setting $t := (t+1) \pmod 2$. The readers, on the other hand, now write no values, only the timestamps. If the reader chooses the writer, it takes the value from the marked field; if it chooses a reader, it takes the value from the unmarked field. Since no observed reader can be more than one write ahead of the actually observed write, this is feasible while maintaining correctness. This results in $O(n)$ replications of the value written resulting in a total of $O(n \log V)$ value bits. This is clearly the optimal order since the writer needs to communicate the value to everyone of the readers through a separate subregister. (Consider a schedule where every reader but one has fallen asleep indefinitely. Wait-freeness requires that the writer writes the value to a subregister being read by the active reader.)

Acknowledgment: I thank Ming Li and Amos Israeli for initial interactions about these ideas in the late eighties. Other interests prevented me from earlier publication. I also thank Grisha Chockler for pointing out a problem with the previous formulation of Lemma 3, and meticulously reading earlier drafts and providing useful comments.

References

1. J. Anderson, Multi-Writer Composite Registers, *Distributed Computing*, 7:4(1994), 175-195.
2. K. Abrahamson, On achieving consensus using a shared memory. In *Proc. 7th ACM Symp. Principles Distribut. Comput.*, 1988, 291–302.
3. B. Bloom, Constructing two-writer atomic registers. *IEEE Trans. on Computers*, 37(1988), 1506–1514.
4. J.E. Burns and G.L. Peterson, Constructing multi-reader atomic values from non-atomic values. In *Proc. 6th ACM Symp. Principles of Distributed Computing.* 1987, 222–231.
5. D. Dolev and N. Shavit, Bounded concurrent time-stamp systems are constructible. *Siam J. Computing*, 26:2(1997), 418-455.
6. C. Dwork and O. Waarts, Simple and efficient bounded concurrent timestamping and the traceable use abstraction. *J. Assoc. Comput. Mach.*, 46:5(1999), 633–666.
7. C. Dwork, M. Herlihy, S. Plotkin and O. Waarts, Time-Lapse snapshots. *SIAM J. Computing*, 28:5(1999), 1848–1874.

8. R. Gawlick, N.A. Lynch, and N. Shavit, Concurrent timestamping made simple. In *Proc. Israeli Symp. Theory Comput. and Systems*, LNCS 601, Springer-Verlag, 1992, 171–183.
9. S. Haldar and K. Vidyasankar, Counterexamples to a one writer multireader atomic shared variable construction of Burns and Peterson. *ACM Oper. Syst. Rev.*, 26:1(1992), 87–88.
10. S. Haldar and K. Vidyasankar, Constructing 1-writer multireader multivalued atomic variables from regular variables. *J. Assoc. Comput. Mach.*, 42:1(1995), 186–203.
11. S. Haldar and K. Vidyasankar, Buffer-optimal constructions of 1-writer multireader multivalued atomic shared variables. *J. Parallel Distr. Comput.*, 31:2(1995), 174–180.
12. S. Haldar and K. Vidyasankar, Simple extensions of 1-writer atomic variable constructions to multiwriter ones. *Acta Informatica*, 33:2(1996), 177–202.
13. S. Haldar and P. Vitanyi, Bounded concurrent timestamp systemss using vector clocks, *J. Assoc. Comp. Mach.*, 49:1(2002), 101-126.
14. M. Herlihy and J. Wing, Linearizability: A correctness condition for concurrent objects. *ACM Trans. Progr. Lang. Syst.*, 12:3(1990), 463–492.
15. A. Israeli and M. Li, Bounded time-stamps. *Distributed Computing*, 6(1993), 205–209.
16. A. Israeli and M. Pinhasov, A concurrent time-stamp scheme which is linear in time and space. In *Proc. 6th Intn'l Workshop Distribut. Alg.*, LNCS 647, Springer-Verlag, Berlin, 1992, 95–109.
17. A. Israeli and A. Shaham, Optimal multi-writer multireader atomic register. In *Proc. 11th ACM Symp. Principles. Distr. Comput.*, 1992, 71–82.
18. L.M. Kirousis, E. Kranakis, and P.M.B. Vitányi, Atomic multireader register. In *Proc. 2nd Intn'l Workshop Distr. Alg.*. LNCS 312, Springer-Velag, Berlin, 1987, 278–296.
19. L. Lamport, On interprocess communication — Part I: Basic formalism, Part II: Algorithms. *Distributed Computing*, 1:2(1986), 77–101.
20. M. Li and P.M.B. Vitányi, Optimality of wait-free atomic multiwriter variables, *Inform. Process. Lett.*, 43:2(1992), 107–112.
21. M. Li, J.T. Tromp, and P.M.B. Vitányi, How to share concurrent wait-free variables. *J. Assoc. Comput. Mach.*, 43:4(1996), 723-746.
22. N. Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1997.
23. G.L. Peterson, Concurrent reading while writing. *ACM Trans. Progr. Lang. Systems*, 5:1(1983), 56–65.
24. G.L. Peterson and J.E. Burns, Concurrent reading while writing II: The multiwriter case. In *Proc. 28th IEEE Symp. Found. Comput. Sci.*, 1987, 383–392.
25. R. Schaffer, On the correctness of atomic multiwriter registers. Report MIT/LCS/TM-364, 1988.
26. A.K. Singh, J.H. Anderson and M.G. Gouda, The elusive atomic register. *J. Assoc. Comput. Mach.*, 41:2(1994), 311-339.
27. R. Newman-Wolfe, A protocol for wait-free, atomic, multi-reader shared variables. In *Proc. 6th ACM Symp. Princples Distribut. Comput.*, 1987, 232–248.
28. P.M.B. Vitányi and B. Awerbuch, Atomic shared register access by asynchronous hardware. In *Proc. 27th IEEE Symp. Found. Comput. Sci.*, 1986, 233–243. Errata: *Ibid.*, 1987, 487.