How to Share Concurrent Asynchronous Wait-Free Variables
(Preliminary Version)

Ming Li

Computer Science Department, York University
Toronto, Ontario M3J 1P3, Canada.

Paul M.B. Vitanyi

Centrum voor Wiskunde en Informatica
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
and
Faculteit Wiskunde en Informatica, Universiteit van Amsterdam

ABSTRACT

We use a structured top-down approach to develop algorithms for atomic variables shared by concurrent asynchronous wait-free processes, starting from the problem specification. By this design we obtain a better understanding of what the algorithms do, why they do it, and that they correctly implement the specification. Our main construction of a multiwriter variable directly from 1-writer 1-reader variables is the first such construction. Simplifications yield multireader algorithms and multiwriter algorithms. The complexity improves that of known algorithms, in the cases where there were any. Our algorithms are timestamp based. We use a new "shooting" technique to recycle used timestamps.

1. Introduction

Lamport [La] has shown how an atomic variable (or register), shared between one writer and one reader, acting asynchronously and without waiting, can be constructed from lower level hardware rather than just assuming its existence. These ideas have aroused interest in the construction of multi-user atomic variables of that type. In a short time this has already lead to a large number of conceptually

* M. Li was supported in part by the National Science Foundation under Grant DCR-8606366 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 an NSERC operating grant at York. Email: li@yuyetti.bitnet, paulv@cwi.nl, paulv@mcvax.bitnet.

extremely complicated ad hoc constructs and (erroneous) proofs. In this paper our goal is to supply a solution to all main problems in the area, left after Lamport's work. We do this by deriving the implementations by correctness-preserving transformations from a higher level specification. As a consequence of their structured programming genesis, the resulting algorithms are correct and easily understood. The final algorithms complexity-wise improve existing algorithms or best combinations of existing algorithms.

The present manuscript has gone through a series of transformations from the preprint [LV]. We present, informally but in some detail, a version of the construction which we think can be explained in a limited number of pages. However, in such a difficult subject (the whoes and sorrows of which will be apparent from the section on 'related work'), in the end one wants a rigorous approach and, preferably, an actual computer implementation. This is provided in the final version [LTV], based on an improved algorithm (cf. Acknowledgement), which is available on request.

1.1. Informal Problem Statement and Main Result

Usually, with asynchronous readers and writers, atomicity of operation executions 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 maximal amount of parallelism inherent in concurrent actions by avoiding waiting altogether in our algorithms. In such a setting, serializability is *not* actively enforced, rather it is the result of a pre-established harmony in the way the execution of the algorithm by the various processors interact. Any of the references, say [La] or [VA], describes the problem area in some detail.

Our point of departure is the solution [Pe, La] of the following problem. (We keep the discussion informal.) A flip-flop is a Boolean variable that can be read (tested) by one processor and written (set and reset) by one other processor. Suppose, we are given atomic flip-flops as building blocks, and are asked to implement an atomic variable with range 1 to n, that can be written by one processor and read by another one. Of course, $\log n$ flip-flops suffice to hold such a value. However, the two processors are asynchronous. Suppose the writer gets stuck after it has set half the bits of the new value. If the reader executes a read after this, it obtains a value that consists of half the new value and half the old one. Obviously, this violates atomicity.

At the outset we state our main result:

Theorem 1. *We supply a construction for an atomic n-reader n-writer variable from $O(n^2)$ atomic 1-reader 1-writer variables, using $O(n)$ accesses of subvariables per operation execution and $O(n)$ control bits per subvariable.*

1.2. Comparison with Related Work.

Related ad hoc and very difficult constructions have been proposed by [SAG,KKV,BP,NW,IL] (for the 1-reader 1-writer to multi-reader 1-writer case) and by [VA,PB,IL,DS] (for the multi-reader 1-writer to the multi-reader multi-writer case). We note that especially the latter construction has appeared to be quite difficult. Both algorithms that have been completely published, and subjected to scrutiny, turned out to contain errors. I.e., the algorithm in [VA] presented in FOCS86 is not fully atomic but only satisfies the weaker "regularity" condition, as pointed out in FOCS87 errata. A modification of this algorithm presented subsequently in FOCS87 [PB], was found to contain several errors by Russel Schaffer [Sc]. The multiwriter algorithm promised in [IL] has not yet been published in any detail. The recent [DS] starts from multi-reader variables, and uses a simplified version of the unbounded tag algorithm of [VA] (as presented in [LV]) as point of departure. Generally, papers in the area are hard to comprehend and check.

With these difficulties, there has been no previous attempt to implement an n-writer n-reader variable directly from 1-reader 1-writer variables, like we present here. Yet we believe the construction we present is relatively simple and transparent. Both problems above, that have been subject of other investigations, are solved by simplifications (as it were 'projections') of our main solution. We appear to improve all existing algorithms under some natural measures of complexity. Worst case complexity comparison of our direct solution here with the best combinations:

paper	control bits	atomic accesses
[This paper]	$O(n^3)$	$O(n)$
[BP]+[PB, Sc]	$\Omega(n^3)$	$\Omega(n^3)$
[SAG, KKV, BP,IL,VA]	$\Omega(n^3)$	$\Omega(n^2)$
[SAG, KKV, BP,DS]	$\Omega(n^3)$	$\Omega(n^2 \log n)$

Explanation: The compound variable (here n-reader n-writer) is composed from primitive variables (here 1-reader 1-writer). To store a value in the compound variable, it is stored in a subset of the primitive variables, along with some control information (like time-stamps) to allow selection of the most recent value. The 'control bits' column displays to over-all number of bits of control information required, summed over all primitive variables ("space" complexity). The 'atomic access' column displays the number of reads/writes from/to primitive variables required in the execution of one read/write from/to the compound variable ("time" complexity). The related work is [Pe], [La], [B], [VA], [AGS], [Ly], [KKV], [PB], [BP], [AKKV], [IL], [NW], [Vi], [LTV], [DS].

2. Definitions, Problem, and Specification

We are primarily interested in the algorithmics of the subject matter, and hence give the basics in a nutshell - as rigorous as we think is needed here. For more formal treatment see [La] and [AKKV]; [Bl], [He] are examples in the area using I/O automata [LT]. A concurrent system consists of a collection of sequential processes that communicate through shared datastructures. The most basic such datastructure is a shared variable. A user of a shared variable V can start an operation execution a (read or write) at any time when it is not engaged in another operation execution, by issuing an "execute a" input command on V. It finishes at some later time when it receives a response from V that the execution of a is completed. We can express the semantics in terms of local value v of a process P and the global value contained in V. In absence of any other concurrent action the result of process P writing its local value v to V is that $V:=v$ is executed, and the result of process P reading the global V is that $v:=V$ is executed.

An *implementation* of V consists of a set of *protocols*, one for each reader and writer process, and a set of shared variables X, Y,..., Z. (A shared variable is sometimes called a *register*.) An operation execution a by user process P on V consists of an execution of the associated protocol which

- starts at time $s(a)$ (the start time)
- applies some transformations on shared and local variables $X, Y,...,Z$
- returns a result to P at time $f(a)>s(a)$ (the finish time).

The start and finish times of all operation executions are supposed to be disjoint. I.e., for operation executions $a \neq b$ we have $s(a) \neq s(b)$, $s(a) \neq f(b)$, and $f(a) \neq f(b)$. All interactions between processes and variables are asynchronous but reliable, and can be thought of as being mediated by a concurrent scheduler atomaton. The read/write operations are total, i.e., they are defined for every state of the variable. An implementation is *wait-free* if:

- the number of sub-variable accesses in an operation execution is bounded by a constant, which depends on the number of readers and writers;
- there is no conditional waiting; i.e., each sub-variable access that has no outstanding sub-sub-variable accesses, has an enabled return transition.

Linearizability or atomicity is defined in terms of equivalence with a sequential system in which interactions are mediated by a sequential scheduler that permits only one operation to execute at any variable at a time. A shared variable is *atomic*, if each read and write of it actually happens, or appears to take effect, instantaneously at some point between its invocation and response, irrespective of its actual duration. This can be formalized as follows.

Let V be a shared variable with associated user processes $P,Q,..,R$ which execute a set of operation executions A on V. Order the set A (of reads and writes) such that action a precedes action b, $a \rightarrow b$, if $f(a)<s(b)$. Note that with this definition $\rightarrow$ is a special type of partial order called an *interval* order (that is, a

transitive binary relation such that if $a \to b$ and $c \to d$ then $a \to d$ or $c \to b$). Define the reading mapping π as a mapping from reads to writes by: if r is a read that returns the value writen by write w, then $\pi(r) = w$. We call the triple $\sigma = (A, \to, \pi)$ a *system execution*.

Definition. A system execution σ is *atomic* if we can extend $\to$ to a total order $\to'$ such that

(A1) $\pi(r) \to' r$, and

(A2) there is no write w such that $\pi(r) \to' w \to' r$.

That is, the partially ordered set of actions can be linearized while respecting the logical read/write order. A shared variable is *atomic* if each system execution $\sigma = (A, \to, \pi)$ of it is atomic.

2.1. The Problem to be Solved

Our goal is to implement an atomic wait-free shared variable V, with n users that can both read and write V. We implement such a V using atomic variables $R_{i,j}$, $1 \leqslant i,j \leqslant n$ for which i is the only associated writer process and j is the only associated reader process. Since i is the only one who can write to variables $R_{i,1},..,R_{i,n}$, we say it *owns* these variables.

2.2. Specification

While the definition of atomicity is quite clear, we transform it into an equivalent specification, from which we can directly derive our first algorithm that implements V. Viz., partition the actions in A into subsets induced by write actions w. Define the equivalence class of a write action w as $[w] = \{a : a = w$ or a is a read and $\pi(a) = w\}$. The precedence relation $\to$ on the actions in A induces a relation $\ll$ on the set of $[w]$'s as follows: $[w_1] \ll [w_2]$ iff $w_1 \neq w_2$ and there are $a \in [w_1]$ and $b \in [w_2]$ such that $a \to b$. The following lemma is from [AKKV].

Lemma 1. *(A1) and (A2) hold iff $\ll$ is acyclic and not$(r \to \pi(r))$ for any read r.*

Proof. "If". If $\ll$ has no cycles then it can be extended to a total order $<$ on the set of $[w]$'s. Define $\to'$ by:

(i) if $[a] \neq [b]$ then $a \to' b$ iff $[a] < [b]$;

(ii) within each $[w]$ we can topologically sort the elements beginneing with the write (since $r \not\to \pi(r)$), so that $[w] = \{w, r_1,..,r_k\}$ and $r_i \to r_j$ implies $i < j$. Now put $w \to' r_1 \to' ... \to' r_k$.

It is easy to verify that $\to'$ exists, and that it is a total order on A satisfying (A1) and (A2).

"Only if." First assume $r \to \pi(r)$ for some read r. Since $\to'$ extends $\to$, we also have $r \to' \pi(r)$ which contradicts (A1). This shows not$(r \to \pi(r))$. Because of (A1) and (A2), each $[w]$ is a consecutive sequence of actions in the total order $\to'$. It follows that the order $<$ on the $[w]$'s, induced by $\to'$, is total. If $[w_1] \ll [w_2]$ then there are $a_1 \in [w_1]$ and $a_2 \in [w_2]$ such that $a_1 \to w_2$. Since $\to'$ extends $\to$, we

also have $a_1 \rightarrow' a_2$, and thus $[w_1] < [w_2]$. We see that $<$ is an extension of $\ll$. Since $<$ is acyclic, so is $\ll$. □

This way we have found the specification an atomic variable has to satisfy: for each of its system executions $\sigma = (A, \rightarrow, \pi)$

(S1) not$(r \rightarrow \pi(r))$ for any read execution r, and

(S2) the induced relation $\ll$ has no cycles.

3. The Basic Algorithm

Our first approximation of the target algorithm captures the essence of the problem solution apart from the boundedness of the constituent shared variables.* Let V be as in the problem description above. For most implementations of V, condition (S1) is trivially satisfied, since violation of it means that a read execution returns a value before the write of it ever started. This condition will be trivially satisfied by all algorithms we consider, so we mention it no further. How can we satisfy (S2)? We proceed as follows. Let $(T, <)$ be a partially ordered set of *tags*. For each system execution $\sigma = (A, \rightarrow, \pi)$, let $tag: A \rightarrow T$ be a function such that

(T1) $tag(a) = tag(b)$ for $b \in [a]$;

(T2) if $a \rightarrow b$ then $tag(a) \leqslant tag(b)$; and

(T3) if $w_1 \neq w_2$ then $tag(w_1) \neq tag(w_2)$.

(T1) ensures that each $[w]$ has a unique tag. If $[w_1] \ll [w_2]$, then by definition $w_1 \neq w_2$ and there are $a_1 \in [w_1]$ and $a_2 \in [w_2]$ with $a_1 \rightarrow a_2$, so that (T2) gives $tag(w_1) \leqslant tag(w_2)$. By (T3), we have $tag(w_1) \neq tag(w_2)$, which combines to $tag(w_1) < tag(w_2)$. It suffices to devise an algorithm such that for each system execution σ there is a function *tag* satisfying (T1), (T2) and (T3), in which case $\ll$ has no cycles and (S2) is satisfied.

Using unbounded tags, we can implement variable V as follows. (T1) will be satisfied by letting read actions copy the tag of the value they choose to return. (T2) will be satisfied by letting the users maximize over the tags that are visible in their actions. Different writers will choose different tags by making the index of the writer part of the tag, i.e., each tag in T is a pair (t,i), where t is a natural number (a *timestamp*) and i is index between 1 and n of the writer that writes the tag first. The $<$-order on T is the total lexicographical order. Finally, a writer will not use a tag that has been used before, because it chooses its tag greater than the maximum visible tag. Thus, (T3) is also satisfied.

This leads us to the basic Algorithm 1. The shared variable V is composed

---

* In fact, this is essentially the "first solution" in [VA] (the correct one of the two solutions presented), there presented with a different proof. This 'matrix' architecture is used in all later constructions such as [SAG, KKV, BP, IL,N-W], that start from 1-reader 1-writer variables to implement 1-writer n-reader variables. This is the only multiwriter algorithm generally accepted as being correct - but it uses unbounded tags.

from atomic subvariables $R_{i,j}$ which can be written by writer i and read by reader j, $1 \leqslant i, j \leqslant n$. Each $R_{i,j}$ contains fields *value* and *tag*, where the latter has subfields *timestamp* and *index*. The algorithm is initialized by setting all fields of all local and subvariables to 1. This puts the system in a state which appears to have resulted from an initial write by user 1 of value 1 with tag (1,1), followed by successive reads from all other users 2, ..., n, such that these all choose maximum tag := (1, 1).

Each user i ($1 \leqslant i \leqslant n$) has a local variable **newvalue** that, at the start the write invocation, contains the value to be assigned to V. Each user i ($1 \leqslant i \leqslant n$) also has a local variable **value** to which the value of V is assigned by the time a read invocation finishes. In the algorithm, i denotes user i, $1 \leqslant i \leqslant n$. The algorithm is displayed in Figure 1.

i **reads value:** /* **value** := V*/

R1) Read $R_{1,i},..,R_{n,i}$.

R2) Select the lexicographical maximal tag, say (t,p).

R3) Write tag := (t, p) and value := value@(t,p) to $R_{i,1},..,R_{i,n}$, and **value** := value@(t,p).

i **writes newvalue** ($1 \leqslant i \leqslant n$)**:** /* V := **newvalue***/

W1) Read $R_{1,i},..,R_{n,i}$.

W2) Select the lexicographical maximal tag, say (t,p).

W3) Write tag := $(t+1, i)$ and value := **newvalue** to $R_{i,1},..,R_{i,n}$.

Figure 1. Algorithm 1

Lemma 2. *Algorithm 1 implements an atomic wait-free multiwriter variable.*

Proof. Obviously Algorithm 1 is wait-free. We only have to argue atomicity. Let $\sigma=(A, \rightarrow, \pi)$ be a system execution according to Algorithm 1. Condition (T1) is satisfied, because according to the read protocol, for each read execution b in $[a]$ we have $tag(b)=tag(a)$. Condition (T2) is satisfied because, for each pair of elements a,b in A, if $f(a)<s(b)$ then $tag(a) \leqslant tag(b)$. Condition (T3) is satisfied, since if $w_1 \neq w_2$ are two writes, then $tag(w_1) \neq tag(w_2)$ (if the writes are by the same user, then the later one has a higher timestamp, if they are by different users the indexes are different). Because $\ll$ can be isomorphically embedded in the total lexicographic order on T, condition (S2) is satisfied. Condition (S1) is satisfied trivially. □

4. Second Algorithm

The only problem with Algorithm 1 is that $T=N \times \{1,..,n\}$ is infinite. The obvious way to proceed is to refine Algorithm 1 to an Algorithm 2 that induces a function tag' such that for each system execution $\sigma=(A, \rightarrow, \pi)$, if $tag(a)=(t,i)$ then $tag'(a)=(t \bmod C, i)$ for some system constant C. The *only* difficulty with this

scheme is that the old *timestamps* (first coordinates of tags) may be confused with the new ones when looping back. However, if the loop is wide enough then outdated timestamps are seen many times by at least one writer. Similarly, long pending actions are seen many times by at least one writer. We exploit this insight using a "shooting" trick, where for now we assume the existence of operations with certain properties, and only in Algorithm 3 show how these operations are implemented. Each time a writer finishes a write, it "shoots" every other read or write execution it "sees" once. A tag is *dead*, if its associated operation execution is shot at least c times by the same writer, c a large enough system constant. In this way, each out of date timestamp gets eliminated after cn writes. When the "information gathering" step R/W1 of an operation execution starts, very old operation executions have received c shots from at least one writer (certainly those such that cn writes have completed since their start).

In Algorithm 2 we "bracket" the information gathering step R/W1 of an operation execution in Algorithm 1 by an extra preliminary step R/W0 that sets up a target associated with the operation execution, and by an extra test step R/W2.0 that checks the number of times the target has been shot. This way an operation can check in step R/W2.0 if it has been shot $\geq c/2$ times by the *same* writer. If so, it completely "overlaps" some write execution from the $c/2$ shooting ones, say the one-before-last, and can safely "abort", i.e., terminate without executing the remainder of its program (see below). An aborting read execution will report the value of the one-before-last shooting write. Hence, at most $cn/2$ writes complete during step R/W1 of a non-aborting operation execution a. Operation executions that have been shot c times by the same writer are "dead". So at the start of an action a, roughly speaking, all actions b such that there are cn actions d with $s(b) < s(d) < f(d) < s(a)$ are dead already. Hence a nonaborting action a only has to consider alive timestamps that are less than $2cn$ apart. Therefore, a loop of length $C = 4cn$ suffices to identify the largest timestamp in the $<2cn$-size "window." The aborting actions present no problem since they overlap completely one write, as will be explained below.

We define three primitives with, for now, informal semantics:

(i) a **aborts**: Operation execution a terminates without executing the remainder of the program.

(ii) **create**(a): Initialize a new operation a such that a is not shot by the "preceding shots" ("concurrent shots" may or may not shoot a). Such an initialization by user i entails adding a new *operation record* to a stack of operation records which is maintained in each own variable $R_{i,1},..,R_{i,n}$. Such an operation record contains a value and tag field and a "target to shoot at". We assume the last c operation records are saved, older ones are discarded.

(iii) a **shoots** b: Write a shoots operation b once.

Semantics.

(1) Write a can only shoot operation b if $s(b) < f(a)$.

(2) Shooting appears to take effect at some point between its invocation and the response that it has been executed. (We don't care whether this point is different for observations by different users.)
(3) User i executing b can determine whether b has been shot at least s times by writes of some user j, by reading the variable $R_{j,i}$, for $s = 0,..,c$, when it checks in step R/W2.0.
(4) User i executing a can determine whether writes by user j have shot some operation b by user k at least s times by reading $R_{j,i}$ and $R_{k,i}$, for $s = 0,..,c$, when it checks in step R/W2.0.

We define an operation (and its associated timestamp) as **dead** if it is shot at least c times by the same writer. In step R/W2 of the read/write protocol below, a timestamp t is **maximal** in the set of alive timestamps, if the set does not contain $t + i \bmod C$, $1 \leqslant i \leqslant C/2$. **Initialization**: Each subvariable $R_{i,j}$ $(1 \leqslant i,j \leqslant n)$ contains an initial operation record, and nobody is shot yet. Timestamps and values in the operation records in all $R_{i,j}$ are initialized as in Algorithm 1. The resulting algorithm is depicted in Figure 2.

i **reads value:** /* **value** := *V**/

R0) **Create**(a).

R1) Read $R_{1,i},..,R_{n,i}$, to select maximal timestamp in step R2

R2.0)Read $R_{1,i},..,R_{n,i}$;

if operation execution a itself is shot at least $c/2$ times by one and the same writer j
then return the one-before-last **value** in $R_{j,i}$ and **abort** *else*

R2) select the lexicographical **maximal** tag, say (t,p), among the operations from step R1 that are shot $<c$ times by each writer;

R3) Write tag := (t, p) and value := value@(t,p) to $R_{i,1},..,R_{i,n}$, and **value** := value@(t,p).

i **writes newvalue:** /* V := **newvalue***/

W0) **Create**(a).

W1) Read $R_{1,i},..,R_{n,i}$, to select maximal timestamp in step W2.

W2.0)Read $R_{1,i},..,R_{n,i}$;

if operation execution a itself is shot at least $c/2$ times by one and the same writer j
then **abort** *else*

W2) select the lexicographical **maximal** tag, say (t,p), among the operations from step W1 that are shot $<c$ times by each writer;

W3) Write tag := $((t+1) \bmod C, i)$, value := **newvalue**, and **shoot** every operation b seen in step W1, to $R_{i,1},..,R_{i,n}$.

Figure 2. Algorithm 2 (operation execution a).

Lemma 3. *Algorithm 2 implements an atomic multiwriter variable (with*

$C = 4cn$).

Proofsketch. We use X_a to denote the writer or reader which performed operation execution a. Consider a fixed system execution $\sigma_2 = (A, \rightarrow_2, \pi_2)$ of Algorithm 2 with all aborted operation executions deleted. Those aborted operations can be inserted at will after we serialize the other operations since no read returns the value of an aborted write and:

Claim 3.1. In Algorithm 2, if an operation execution aborts, then it overlaps completely a write by the writer that shot it $c/2$ times. Therefore the aborted read/write can be inserted immediately after/before the *one-before-last* such write in the atomic order.

Proofsketch of Claim 3.1. If operation a of processor X_a determines in step R/W2.0 that it is shot $c/2$ times by writer W, then obviously a overlaps the one-before-last shooting write by W completely, for c large enough. If a is a write then it is never read since it aborts, and it is hidden in the atomic order by the write just after it. If a is a read, then it reports the value of the one but last write and is put just after it in the atomic order. Both orderings are consistent with the precedence relations and the logical read-write order. □

Now let Algorithm 1 execute exactly the same schedule of operation executions (without the aborted ones) as Algorithm 2, such that corresponding reads and writes of the $R_{i,j}$'s happen at precisely the same times. (I.e., the atomic subactions in steps R1, W1, R3, W3.) This results in a system execution $\sigma_1 = (A, \rightarrow_1, \pi_1)$, with $\rightarrow_1$ an extension of $\rightarrow_2$ (because the operation executions time intervals in σ_1 are subintervals of the corresponding ones in σ_2.) We have to show that $\pi_1 = \pi_2$. Let $tag(a)$ be the tag of a in Algorithm 1 and let $tag'(a)$ be the tag of a in Algorithm 2.

We show that, if operation execution a in Algorithm 1 selects the maximum tag $tag(max)$ with max in $S_1 = \{b_1,..,b_n\}$, then the corresponding operation execution a in Algorithm 2 selects $tag'(max)$ with max in $S_2 \subseteq S_1$. (I.e., S_2 consists of the "alive" operation executions in steps R2 and W2.) The difference between S_2 and S_1 is the set of killed operation executions. We need to show that max cannot be in $S_1 - S_2$, and that with $tag'(max) = (q,j)$, for all a in S_2 and $tag'(a) = (p,i)$, we have $p \neq (q+r) \bmod C$ for $1 \leq r \leq C/2$. Since the "window" in which the alive timestamps cluster is therefore of size less than half of the "loop", and contains the maximal tag $tag(max)$ chosen by Algorithm 1, it follows that $tag'(max) = tag(max) \bmod C$ is chosen by Algorithm 2. This then implies $\pi_1 = \pi_2$. We delegate the remainder of the proof to the Appendix. □

5. Algorithm 3

We now give an implementation of the shooting primitives to derive Algorithm 3. Written down our solution looks complicated. However, the basic idea is simple and the reader may design another implementation that suits her/him better. (In fact, in [LTV] we opt for a different implementation.) The correctness of Algorithm 3 follows from the correctness of Algorithm 2, by demonstration that the

shooting primitives have the properties (1)-(4) claimed in relation to Algorithm 2. The informally mentioned "operation records" in Algorithm 2 correspond to "frames" below. With each operation execution execution a (a read or a write) we associate a $frame_{[[a]]}$ of the format depicted in Figure 3.

operation a of user i $(1 \leqslant i \leqslant n)$
$DIE^{[[a]]}$
$SHOOT^{[[a]]}$
$(timestamp^{[[a]]}, i)$
$Value^{[[a]]}$

Figure 3. Frame in Algorithm 3

Each register $R_{i,j}$ holds c frames: $frame_{ci-l}$, $l = 0,..,c-1$. When $frame_j$ holds operation a, we denote the *frame index* associated with a by $[[a]] = j$, and we also write $frame_{[[a]]}$. At any fixed time, the system contains up to cn timestamps in distinct frames, the frames indexed by $1,..,cn$. Each frame contains the current status of an operation execution: a tag$=(timestamp,i)$, its *value*, and $DIE^{[[a]]}$, $SHOOT^{[[a]]}$ arrays to implement the recent shooting history.* The semantics and lay-out of the arrays is roughly described by:

$$DIE^{FRAME\ SHOT\ AT}(PROCESS\ THAT\ SHOOTS,\ TARGET\ SHOT\ AT)$$

$$SHOOT^{FRAME\ THAT\ SHOOTS}(FRAME\ SHOT\ AT,\ TARGET\ SHOT\ AT)$$

Since there are n users, each of which can shoot, and each frame will have c targets that can be shot at, the $DIE^{[[a]]}(1{:}n,\ 1{:}c)$'s are $n \times c$ arrays. Similarly, there are cn frames to shoot at, and c targets in each frame, so the $SHOOT^{[[a]]}(1{:}cn,\ 1{:}c)$'s are $cn \times c$ arrays. Each array element is a $(c+1)$-ary digit. We say that operation execution a is *killed* by the operation execution d, the *killer*, iff $DIE^{[[a]]}(\lceil [[[d]]/c \rceil,\ 1{:}c) = SHOOT^{[[d]]}([[a]],\ 1{:}c)$.

Remark. The purpose of the $(c+1)$-ary entries in the arrays is that in a freshly created frame of operation execution a, $[[a]] = j$, each $DIE^j[k,i]$ entry can be set different from the c $SHOOT^{k'}(j,i)$ entries in the c frames k' contained in variables $R_{k,l}$ $(1 \leqslant l \leqslant n$ and $k = \lceil k'/c \rceil)$.

We implement shooting primitives and an "abort" primitive as follows (k is a local $(c+1)$-ary variable of each processor):

---

* The notion we use to implement actions like creation of frames or shooting, can be compared to a hotel switch. A hotel switch is a switch that can be switched from two different locations. I.e., there is a light switch upstairs and downstairs. From both one can switch the light on or off. The light (on or off) is a shared variable between two parties, one at each switch, that can be set and reset by both parties using their own switches (i.e., local variables).

(i) **create**(a): This consists in setting a new $frame_{[[a]]}$ in each $R_{i,j}$, $1 \leqslant j \leqslant n$. Now before we start with this, each $R_{i,j}$ contains the c frames of the most recent operation executions by $i = X_a$. To accommodate $frame_{[[a]]}$, we have to delete another frame in this stack. Let $frame_{[[b]]}$ be the most recent one. To start the new operation a, if b is a read execution, then processor i replaces $frame_{[[b]]}$ by $frame_{[[a]]}$, else it replaces the least recent frame by $frame_{[[a]]}$. Now we describe what the contents of $frame_{[[a]]}$ is going to be. To determine this, i first reads all $SHOOT$ arrays of $R_{1,i},..,R_{n,i}$. Then, in atomic writes to $R_{i,1},..,R_{i,n}$, processor i sets $frame_{[[a]]}$ with EMPTY timestamp (which is less than all other timestamps) field and EMPTY value field, $SHOOT^{[[a]]} := SHOOT^{[[b]]}$ (to inherit the most recent shooting record of writer i) and, using the $SHOOT$ arrays of $R_{1,i},..,R_{n,i}$ read above, set $DIE^{[[a]]}(\lceil [[d]]/c \rceil, i) := \neq SHOOT^{[[d]]}([[a]], i)$ (set itself alive) for $[[d]] = 1,..,cn$, $i = 1,..,c$.

(ii) a **shoots** b: The operation execution a *shoots* b using k by setting $SHOOT^{[[a]]}([[b]], k) := DIE^{[[b]]}([[a]], k)$.

(iii) a **aborts**: operation execution a terminates without further changing any local or register variable.

Initialize: $k = 0$, and DIE, $SHOOT$ arrays such that nobody shoots anybody. Timestamps and values in frames in all R_i are initialized as in Algorithm 2. The maximality criterion in step R/W2 is the same as in Algorithm 2. The resulting algorithm is depicted in Figure 4.

Lemma 4. *Algorithm 3 implements an atomic multiwriter variable with* $C = 4cn$.

Proofsketch. The only thing we have to prove is that this implementation of the shooting primitives have the semantics properties (1)-(4) we required above. Properties (1), (2) and (3) are straightforward, and property (4) is proven in the Appendix. □

6. Algorithm 4

Algorithm 3 satisfies Theorem 1 but for the control bit complexity which is $O(n^2)$. This depends on $c = O(n)$ which we needed only to satisfy Claim 3.2 in the Appendix. However, with a simple expedient it suffices $c = O(1)$, achieving the control bit complexity of Theorem 1. The only change is to execute a complete read extra, according to the read execution of Algorithm 3, inside each write execution, between step $W1$ and step $W2.0$ in Algorithm 3. This 'dummy' read for user i uses an extra row $R_{n+i,1},..,R_{n+i,n}$ to perform its write phases on (steps R0, R3). If the 'dummy' read aborts, then so does the write that spawns it. The 'dummy' read does not need to return a value; its only function is to propagate the maximum tag it has seen by writing it in its row $R_{n+i,1},...,R_{n+i,n}$. Each column read phase in both the read and write protocol for user i now reads $R_{1,i},...,R_{2n,i}$ instead of $R_{1,i},...,R_{n,i}$ (steps R0, R1, R2.0, W0, W1, W2.0, in Algorithm 3). The resulting changes to Algorithm 3 give Algorithm 4. In the analogous proof to that of Algorithm 2, the

i **reads value:** /* **value** := *V**/

R0) **Create**(*a*).

R1) Read $R_{1,i},..,R_{n,i}$, to select maximal timestamp in step R2.

R2.0)Read $R_{1,i},..,R_{n,i}$;

if operation execution *a* itself is shot at least $c/2$ times by some writer *j* *then* return the one-before-last **value** in $R_{j,i}$ and **abort** *else*

R2) select the lexicographical **maximal** tag, say (t,p), among the operations from step R1 that are shot $<c$ times by each writer;

R3) Write tag := (t, p) and value := value@(t,p) to $frame_{[[a]]}$ in $R_{i,1},..,R_{i,n}$ in one atomic write each (without changing the other arrays); **value** := value@(t,p).

i **writes newvalue** ($1 \leqslant i \leqslant n$): /* *V* := **newvalue***/

W0) **Create**(*a*).

W1) Read $R_{1,i},...,R_{n,i}$, to select maximal timestamp in step W2.

W2.0)Read $R_{1,i},...,R_{n,i}$;

if operation execution *a* itself is shot at least $c/2$ times by some writer *j* *then* **abort** *else*

W2) select the lexicographical **maximal** tag, say (t,p), among the operations from step W1 that are shot $<c$ times by each writer;

W3) Write tag := $((t+1) \bmod C, i)$ and value := **newvalue** to tag and value fields in $frame_{[[a]]}$, and **shoot** every other frame read in step W1 using *k* in one atomic write per variable $R_{i,1},..,R_{i,n}$; $k := (k+1) \bmod c$.

Figure 4. Algorithm 3 (operation execution *a*).

reads fill the same role, but each write in Algorithm 4 now corresponds to a combination of a write and a read in Algorithm 1. This reduces the required size of *c* in Claim 3.2 to $O(1)$. This solution, while somewhat clumsy, has the advantage that the proof of correctness of Algorithm 3 carries over immediately to Algorithm 4. But in [LTV] we have found another, more basic, solution. That solution has the added advantage of making the write algorithm a simple extension of the read algorithm, and hence the combined algorithm is shorter.

7. Construction of Multiwriter Variables from Multireader Variables

A simplification of Algorithm 3 corresponds to the problem addressed in [VA, PB, IL, DS]. Viz., to implement an atomic wait-free shared variable *W*, with *n* *n*-reader single writer atomic sub-variables 1,..,*n*.

To implement *W*, collapse each row $R_{i,1},..,R_{i,n}$ in Algorithms 1-3 to the single multireader variable R_i owned by *i*. So the multiple atomic writes in steps R/W0 and R/W3 of the algorithms turn into a single atomic write. Each variable $R_{j,i}$ ($1 \leqslant j \leqslant n$), read in steps R/W1 and R/W2.0 is replaced by R_j. As a consequence of the fact that each columns $R_{1,i},..,R_{n,i}$ is reduced to the same column $R_1,..,R_n$, the constant *c* can be reduced from $O(n)$ to $O(1)$ outright, without having to use the trick of Algorithm 4. Moreover, in a read the users do not have to execute the write

in step R/W3. This solution uses $O(n)$ control bits per sub-variable R_i and $O(n)$ atomic accesses of sub-variables R_i per read or write of W. Explicit algorithms are given in [LTV], but it is a simple exercise to derive them as indicated.

8. Construction of Multireader Variables from Singlereader Variables

Another simplification of Algorithm 3 corresponds to the problem that addressed in [SAG],[KKV],[BP],[NW],[IL]. Viz., to implement an atomic wait-free shared variable R, with user 1 the only writer, and users $1,..,n$ the readers. R is implemented using atomic 1-reader 1-writer variables $R_{i,j}$, $1 \leqslant i,j \leqslant n$ for which user i is the only associated writer, and user j is the only associated reader. We show how to implement R using $O(n)$ control bits in each variable $R_{1,j}$ $(1 \leqslant j \leqslant n)$ owned by the writer 1, and $O(1)$ control bits in each variable $R_{i,j}$ $(1 \leqslant j < i \leqslant n)$ owned by readers $2,..,n$. (Total, $O(n^2)$ control bits.) The construction is the same as Algorithms 1-3, except

(1) just like in the previous simplified solution, system constant c is $O(1)$; and

(2) the writer now only needs to maintain *SHOOT* arrays (and no *DIE* arrays), and as before the readers need to maintain only *DIE* arrays (and no *SHOOT* arrays). Hence, the *SHOOT* arrays of the writer have $O(n)$ bits, but the *DIE* arrays of the readers need only $O(1)$ bits; and

(3) obviously the write will never abort.

This brings the total number of bits needed to $O(n^2)$, which is optimal. The number of atomic accesses of $R_{i,j}$'s in each read or write is $O(n)$, which is optimal too. Explicit algorithms are given in [LTV], but it is a simple exercise to derive them as indicated.

9. On Optimality of Control Bits for Multiwriter Constructions

Is the linear tag-size optimal? In [IL], an $\Omega(n)$ lower bound is proved for the tag-size for sequential binary comparison algorithms. Let us explain what this means in the current context. An algorithm is *sequential*, if it contains no overlapping operation executions. The algorithms we have considered are *concurrent*, they allow overlapping. A lower bound proven for a sequential restriction of an algorithm holds *a fortiori* for the concurrent version. In our context *binary comparison* means that a user can determine the (apparent) atomic order between every two writes. However, it does not need to do so - we need only to be able to determine the *latest* write from a set of writes, and we do not care about the relative order among the remaining writes. In fact, the lower bound proven in [IL] is not relevant for the multiwriter problem, since we exhibit an $O(\log n)$ upper bound for a sequential solution below.

We generalize the time-stamp system defined in [IL], removing all restrictions: This discussion assumes some knowledge of [IL]. A (sequential) *generalized time-stamp system of order n* is $<G,f>$, where G is a set of nodes (or just numbers) and f is a symmetric function from G^n to G such that the following n pebble game can

be infinitely played on G,

(1) Initially all n pebbles are on the first node;

(2) At each step, the adversary chooses a pebble and the pebbler has to move this pebble to a node v such that $f(\{v, v_1, \cdots, v_{n-1}\}) = v$ where $v_1, \cdots, v_{n-1}$ are the nodes in G where the rest of the $n-1$ pebbles are located.

We call f the labeling function. Obviously the new time-stamp system has most nice properties of the old time-stamp system of [IL]. However in [IL] it was proved that $\Omega(2^n)$ nodes are needed for the sequential [IL]-time-stamp system of order n. The following modification of Algorithm 1, assuming the operation executions do not overlap, needs only $O(\log n)$ bits to encode a (sequential) generalized time-stamp system of order n. The *tags* (= time-stamps in this case) are just $1,...,n$ and are initialized with value 1. The algorithm is displayed in Figure 5.

i **reads value:** /* **value** $:=$ V*/

R1) Read $R_{1,i},..,R_{n,i}$.

R2) Compute $m = (\sum_j tag@R_{j,i}) \bmod n$.

R3) **value** $:=$ value@$R_{m,i}$.

i **writes newvalue** $(1 \leqslant i \leqslant n)$: /* V $:=$ **newvalue***/

W1) Read $R_{1,i},..,R_{n,i}$.

W2) Compute m such that $i = (m + \sum_{j \neq i} tag@R_{j,i}) \bmod n$.

W3) Write $tag := m$ and value $:=$ **newvalue** to $R_{i,1},..,R_{i,n}$.

Figure 5. Sequential multiwriter algorithm using n tags.

Appendix

Remainder Proofsketch of Lemma 3. The lemma follows immediately from Claim 3.3 below. In the proof of Claim 3.3 we need the following:

Claim 3.2. In Algorithm 2, if a non-aborting write a is shot for the cth time by X_b in write b, then $tag(b) > tag(a)$ in Algorithm 1.

Proofsketch of Claim 3.2. Since write a did not abort at the end of step W2.0, a is shot $< c/2$ times by X_b at time τ_2, the time when step W2.0 of a starts. In order for X_b to kill a, it needs to shoot a at least $> c/2$ more times. The second such shot can finish only after a finishes its step W1. Hence the next shots have to come from complete writes that start after a has already scanned all timestamps from which it is going to select the maximal one. Now suppose that $tag(e) = (t, i)$ is the highest tag a has read in step W1 of Algorithm 1, while e is not completed yet. Then, $tag(a) = (t+1, \cdot)$. However, because of a chain of operation executions that have partially completed step W3, t may be larger than t' where (t', X_b) is the tag of a write execution that shoots a. But a little reflection shows that at any fixed time, if (t_p, r) is the highest tag in column $R_{1,p},..,R_{n,p}$ and (t_q, s) is the highest tag in column $R_{1,q},..,R_{n,q}$ then $|t_p - t_q| < n$. Therefore, at time τ_2, the maximum tag $tag(f) = (t', j)$ in any column $R_{1,k},..,R_{n,k}$, must satisfy $t' + n > t$. Now each subsequent nonaborting write of X_b writes a larger timestamp than the previous one. Therefore, with

$c/2 > n+1$,

the last of the $c/2$ writes by X_b that shoot *a after* time τ_2, say write f, has $tag(f)=(t'', X_b)$, with $t'' \geq t'+n+1 > t+1$, which proves the claim. □

The lemma then follows from the following claim:

Claim 3.3. Let $R = a_1, a_2, \ldots$ be a total atomic ordered set of nonaborting operation executions associated with Algorithm 1. In both Algorithms 1 and 2, for all i, a_i selects the tag associated with the same write in corresponding steps W/R2 in Algorithms 1 and 2.

Proofsketch of Claim 3.3. By induction on a_i's.

Base. Both protocols are identically initialized.

Induction. Assume that the claim is true for $i = 1, \cdots, k-1$. Consider action a_k (which did not abort) in Algorithms 1 and 2.

(i) We first establish that if $tag'(max)$ is the maximal tag selected by a_k in Algorithm 1, then max is not killed (shot for the cth time) by writer X_{a_l} in write a_l and a_k scans both $tag'(max)$ and $tag'(a_l)$ in step R/W1 of Algorithm 2. Namely, as before let $S_i =$ set of (alive) tags that were obtained by a_k in step R/W1 of Algorithm i in system executions σ_i, for $i = 1,2$ respectively. We exploit the assumption that all atomic subactions of the steps R/W1 and R/W3 of the corresponding operation executions in the system executions σ_1 and σ_2 according to Algorithms 1 and 2 take place at exactly the same times. In particular therefore, corresponding operation executions scan tags from the corresponding same sets of writes in step R/W1 of their Algorithm, in both system executions. Let $tag(max)$ be the maximal tag in S_1. We know that max is not killed by any operation execution a_l with $a_l \in S_2$ in the system execution according to Algorithm 2, since otherwise the killer's tag $tag(a_l)$ would also be in S_1 and by Claim 3.2, $tag(a_l) > tag(max)$ in Algorithm 1, a contradiction.

(ii) We secondly establish that all alive timestamps scanned by a_k in Algorithm 2 are in a small enough window. We first observe that at any time instant all alive timestamps are within an interval of size cn in the mod C cycle, say operation execution a least and b greatest in the total order R. Namely, if a and b are $\geq cn$ apart, then there are at least cn complete writes within the time interval from the start of a until the finish of b. Consequently, one writer executes at least c writes of the cn, shooting a at least c times, and killing it: contradiction.

Let step R/W0 finish at time τ_0, and step R/W2.0 start at time τ_2. At time τ_0, all alive timestamps written in own variables are within an interval of size cn by the above argument. From τ_0 to τ_2, less than $cn/2$ consecutive timestamps can be written. If not, then some writer would shoot a_k twice and a_k would detect this after τ_2 in its scan in step R/W2.0 and abort, contradicting the assumption that a_k did not abort. Therefore the alive timestamps observed by a_k are clustered in a window of size $\leq 3cn/2$ which include the $tag(max)$ seen by a_k according to the system execution σ_1, which corresponds to the maximal tag in the $<2cn$-size "window" among the alive timestamps seen by a_k in σ_2. So Algorithms 1 and 2 choose the tags corresponding to a same write in a_k. This finishes the induction. □

Remainder Proofsketch of Lemma 4. Property (4) follows by: Assume that writer X_b shoots e for the ith time in write b, $i \leq c$. By Algorithm 3, e never changes its *DIE* arrays once initialized. X_b will keep the shooting bits for e in its most recent frame, as long as it scans e in step W1 of a nonaborting write execution. (Actually X_b keeps shooting at e at every write, and a read does not change shooting bits). Hence once e got shot $\geq i$ times by

X_b, $i=0,..,c$, that fact will not change until $frame_{[[e]]}$ is replaced by a fresh frame of a new operation by X_e. After $frame_{[[e]]}$ is removed by X_e, it takes at least 1 concurrent write, $c-2$ complete writes, and the **create** part of a cth write by X_b, to "fade out" the currently most recent frame and its *SHOOT* array containing the "killing bits" for e. Namely, X_b keeps c frames and it replaces the older frames first. A first concurrent write of X_b can change the *SHOOT* array in the current frame, the $c-2$ complete writes by X_b replace the $c-2$ oldest frames, and the **create** part of the cth write by X_b modifies the *SHOOT* array of the remaining frame. In case X_b reads, the shooting bits are not changed at all. Suppose X_b has shot e at least i times. Now in step R/W1 of Algorithm 3, if a sees both $frame_{[[e]]}$ and i killing bits in a *SHOOT* array of X_b, then a concludes that e is shot i times. If a only sees $frame_{[[e]]}$ but did not see i killing bits in the *SHOOT* array of any of the c frames of X_b, then X_b has written at least $c-1$ writes, at least $c-2$ of which complete, after $frame_{[[e]]}$ is removed and hence after a sees $frame_{[[e]]}$. Therefore, X_b shoots a at least $c-2$ times before τ_2 and a must detect this and abort at step R/W2.0 of Algorithm 3 since $c-2>c/2$ for large enough c: contradiction □

Acknowledgements

We thank John Tromp for helping with a rigorous presentation based on the method presented in this paper. His involvement resulted in several improvements. Moreover, he implemented the new algorithms in C. By extensive testing simulated asynchronous runs, several candidate algorithms were eliminated, and parameters were optimized. The elegant result, with unambiguous low-level code algorithms and rigorous proofs of correctness, is reported in [LTV], which forms the definitive version of this paper. We thank Amos Israeli for discussions on an early version of the present paper.

References

[AKKV] B. Awerbuch, L. Kirousis, E. Kranakis and P.M.B. Vitányi, "A proof technique for register atomicity", In: Proc. 8th Conf. Found. Software Techn. & Theoret. Comp. Sci., Lecture Notes in Computer Science, Vol. 338, pp. 286-303, Springer Verlag, 1988.

[Bl] B. Bloom, "Constructing Two-writer Atomic Registers," IEEE Transactions on Computers, **37**(1988), pp. 249-259

[BP] J.E. Burns and G.L. Peterson, "Constructing Multi-reader Atomic Values From Non-atomic Values", Proc. 6th ACM Symp. on Principles of Distributed Computing, pp. 222-231, 1987.

[DS] D. Dolev and N. Shavit, Bounded concurrent time-stamp systems are constructible, Extended Abstract, January 9, 1989. (To appear in STOC-89.)

[He] M.P. Herlihy, "Impossibility and Universality Results for Wait-Free Synchronization", Proc. PODC, 1988.

[IL] A. Israeli and M. Li, "Bounded Time-Stamps", Proc. 28th IEEE Symp. on Foundations of Computer Science, pp. 371-382, 1987.

[LTV] M. Li, J. Tromp, and P.M.B. Vitányi, "How to share concurrent wait-free variables", Centre for Mathematics and Computer Science, Amsterdam, March 1989, submitted.

[LV] M. Li, P.M.B. Vitányi, "A very simple construction for atomic multiwriter register", Techn. Rept. TR-01-87, Aiken Computation Laboratory, Harvard University, November 1987.

[KKV] L.M. Kirousis, E. Kranakis, P.M.B. Vitányi, "Atomic Multireader Register", Proc. 2nd International Workshop on Distributed Computing, Amsterdam, J. van Leeuwen (ed.), Lecture Notes in Computer Science, Vol. 312, pp. 278-296, 1987.

[La] L. Lamport, "On Interprocess Communication Parts I and II", Distributed Computing, Vol. 1, 1986, pp. 77-101.

[LT] N. Lynch and M. Tuttle, Hierarchical correctness proofs for distributed algorithms, Proc. 6th ACM Symposium on Principles of Distributed Computing, 1987.

[NW] R. Newman-Wolfe, "A Protocol for Wait-free, Atomic, Multi-Reader Shared Variables", Proc. 6th ACM Symp. on Principles of Distributed Computing, pp. 232-248, 1987.

[PB] G.L. Peterson and J.E. Burns, "Concurrent reading while writing II: the multiwriter case", Proc. 28th IEEE Symp. on Foundations of Computer Science, pp. 383-392, 1987.

[Pe] G.L. Peterson, "Concurrent reading while writing", ACM Transactions on Programming Languages and Systems, vol. 5, No.1, 1983, pp. 46-55.

[Sc] R. Schaffer, On the correctness of atomic multi-writer registers, Tech. Rept. MIT/LCS/TM-364, MIT Lab. for Computer Science, June 1988.

[SAG] A.K. Singh, J.H. Anderson, M.G. Gouda, "The Elusive Atomic Register Revisited", Proc. 6th ACM Symp. on Principles of Distributed Computing, pp. 206-221, 1987.

[Vi] Vidyasankar, Converting Lamport's Regular Register to an atomic register, *Information Processing Letters*, **28**(1988), pp. 287-290.

[VA] P.M.B. Vitányi and B. Awerbuch, "Atomic Shared Register Access by Asynchronous Hardware", Proc. 27th IEEE Symp. on Foundations of Computer Science, pp. 233-243. (Errata, **Ibid.**, 1987.)