Wait-Free Handshaking using Rainbow Coloring

Marina Papatriantafilou and Philippas Tsigas

Computing Sciences Department, Chalmers University of Technology and Göteborg University, S-412 96 Göteborg
Email: {ptrianta,tsigas}@cs.chalmers.se

The construction of shared data objects is a fundamental issue in asynchronous concurrent systems, since these objects provide the means for communication and synchronization between processes. Constructions which guarantee that concurrent access to the shared object by processes is free from waiting are of particular interest, since they help to increase the amount of parallelism and to provide fault-tolerance. The problem of constructing a k-valued wait-free shared register out of binary subregisters of the same type where each write access consists of one subwrite (*constructions with one-write*) is important, since it lies at the heart of studying lower bounds of the complexities of register constructions and trade-offs between them. The first such construction was for the safe register case; it uses k binary safe registers and exploits the properties of a rainbow coloring function of a hypercube graph. The best known construction for the regular (atomic) case uses $\binom{k}{2}$ binary regular (resp. atomic) registers, while if the one-write requirement is lifted, there exists a construction that uses $4(\log k + 1)$ binary registers. Here we show how rainbow coloring can be extended to simulate handshaking between the reader and the writer of the register, thus offering a wait-free solution for the atomic case with one reader, using only $3k - 2$ binary registers. The best known lower bound for such a construction is $k - 1$.

Received month date, year; revised month date, year

1. INTRODUCTION

Background

In all forms of communication in a concurrent system the problem of sharing data between multiple processes must be faced at some level. The traditional way to share data among processes which either read or write the data is to require either mutual exclusion or that a write have exclusive access to the data, thus making only concurrent reading possible [1]. The requirement that some actions happen in an exclusive manner implies waiting by some process for another. However, in an asynchronous system, where some processors may be inherently faster than others, the above approach would slow a fast process down to the speed of a slow one.

A natural property to require from an implementation of a shared data object in an asynchronous concurrent system is to guarantee that any process can complete any access to the object in a finite number of steps, regardless of the execution speeds of the other processes. Such an implementation is called *wait-free*. Wait-free shared data objects not only help in taking advantage of the inherent parallelism in concurrent systems, but also guarantee resiliency to halting/stopping failures, since a process that crashes while accessing the object cannot block the progress of any other process intending to access the same object.

A shared variable that supports concurrent read and write operations in a wait-free manner is also called a *wait-free shared register*; from this point in the paper, we adopt the convention to call it *register*.

A register is characterized by the number of readers that may concurrently read it, the number of writers that may concurrently write it, as well as the number of values it can take on. Registers are also classified according to the consistency guarantees they provide in the presence of concurrent operations. Three kinds of consistency guarantees, namely *safeness*, *regularity* and *atomicity*, have been defined by Lamport in [2] and have become of fundamental importance in the study of shared registers. According to those definitions, a shared register, which can be concurrently accessed by one writer process and one or more reading processes, is called:

safe if it guarantees only that a read which does not happen concurrently with any write always returns the most recent value written to the register. The safeness property ensures nothing for the value returned by a read which overlaps with writes; it may equal any possible value of the register;

regular if, besides ensuring safeness, it guarantees that a read that happens concurrently with one or more writes returns a "reasonable" value, which might be either the old one or one of the values written by one of the overlapping writes;

atomic if it guarantees that even when read and write operations overlap, there exists a way to "shrink" each one of them in an atomic grain of time which lies in its respective time duration, in a way that the value returned by each read equals the value written by the most recent write according to the sequence of "shrunk" operations in the time axis.

These dimensions imply a hierarchy on registers. The idea is to start with simple communication primitives (such as single-writer single-reader safe registers), which can be provided directly in hardware, and successively construct more powerful multi-reader (even multi-writer) multi-valued objects [2, 3]. This procedure leads to modular system organization.

Contribution of this paper and related work

Despite the fact that there has been a great deal of research on implementations of stronger registers out of weaker ones [4, 5, 6, 7, 8, 9, 2, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20], to the best of our knowledge, comparatively few results have appeared studying the necessary costs incurred by such implementations [21, 22, 23, 16].

Chaudhuri and Welch in [22] summarize the issues involved in the study of the intrinsic complexity of register constructions: Since registers may differ in several dimensions, the inherent complexity of constructing a strong register out of weaker ones has multiple cases to be examined. They focus on two parameters of interest: the number of values and the consistency guarantees of the register. Thus, they propose the problem of studying the inherent cost of constructing multireader, single-writer, k-valued safe, regular and atomic registers out of multireader, single-writer, 2-valued (binary) safe, regular and atomic registers, respectively. Following their abbreviation, we refer to such constructions as *k-valued from 2-valued safe/regular/atomic* register constructions. The cost measures are the number of binary registers used and the number of subreads and subwrites performed during each read or write operation of the registers. In that paper the particular classes examined are those of k-valued from 2-valued safe and regular constructions. First they prove that for the case in which the writer performs only one write suboperation, $k-1$ subregisters are necessary. As a second step they give an algorithm which implements a safe register, using as an encoding function a special function that can vertex-color a $(k-1)$-dimensional hypercube (recalling its definition: a *n-dimensional hypercube* is a regular undirected graph with 2^n vertices labelled from 0 to 2^n-1, where two vertices are connected if the binary representation of their labels differ in exactly one bit; figure 1 shows a 3-dimensional hypercube) with k colors, after proving that such a coloring exists if and only if k is a power of 2.

Kant and van Leeuwen in [24] independently have shown the same result for this special coloring of hypercubes and they applied it to the file distribution problem.

The aforementioned special coloring of a $(k-1)$-dimensional hypercube with k colors is called *rainbow coloring* and is such that each node of the hypercube has exactly one neighbor with each one of the $k-1$ colors other than its own.

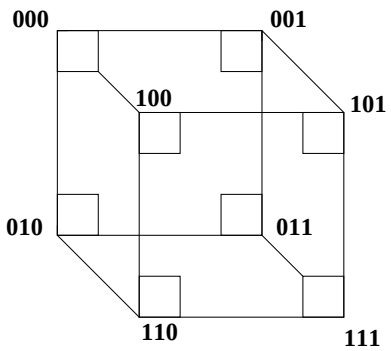

FIGURE 1. A 3-dimensional hypercube

In a subsequent paper [21], Chaudhuri, Kosa and Welch give a construction of a k-valued regular (atomic) register out of binary regular (resp. atomic) ones in which each write performs only one subwrite to one of them and which requires $\binom{k}{2}$ subregisters. Both these algorithms in [22] and in [21] are for the multi-reader case while they have the writer perform only one suboperation per write. If the one-write requirement is lifted, there exists a construction that uses $4(\log k+1)$ binary registers [16].

Here we show how a rainbow coloring —which has been used to construct the weakest type of registers, the safe one— can be extended to implement a powerful *handshaking* mechanism. The handshaking mechanism is particularly important in wait-free algorithms, as has been pointed out by Tromp in [16], by Kirousis, Spirakis and Tsigas in [25] and by Dwork et al. in [26, 27]. Roughly speaking, handshaking involves having the processes play a "hide-and-seek" game by accessing different physical locations (shared variables), so as to minimize concurrent access of the same memory locations. In section 3, the mechanism is described in more detail. Using our implementation, we get a k-valued from 2-valued atomic construction where the writer performs one write suboperation per operation and there are no overlapping reads. Our construction uses $3\cdot 2^{\lceil \log k\rceil}-2$ subregisters (when k is a power of 2 this is $3k-2$), while the best known lower bound for such a construction is $k-1$.

Before proceeding to the description of our construc-

tion, the following section presents more precisely the requirements of the problem.

2. PRELIMINARIES

A *construction* of a register comprises of a data structure consisting of memory cells called *subregisters*, a set of initial values for the subregisters and a set of read and write procedures (aka a *protocol*) which provide the means to the processes to access the register.

When a process needs to perform either a read or a write operation on the register it must invoke the respective procedure; we call this process either *reader* or *writer*, respectively. Each *procedure execution*, or shortly *operation*, is a sequential execution of a procedure's statements (*steps*), which may be either read or write suboperations on the subregisters or some local computations of the procedure. To avoid confusion between operations on the constructed register and operations on the subregisters used in the construction, the term operations is used only for the former and *suboperations* is used for the latter.

A construction $\mathcal{C}$ is called *wait-free* if it guarantees that any operation will complete in a bounded number of steps. The wait-free condition rules out unbounded busy waiting, as well as conditional waiting.

In a global time model [2] each operation q "occupies" a time interval $[s(q), f(q)]$ on one linear time axis $(s(q) < f(q))$; think of $s(q), f(q)$ as the starting and finishing time instants of q. During this time interval the operation is said to be *pending*.

A *precedence relation* among a set of operations (denoted by '$\rightarrow$') is a strict partial order: $q_1 \rightarrow q_2$ means that q_1 ends before q_2 starts. The precedence relation is extended to relate suboperations of operations that are related: if $q_1 \rightarrow q_2$, then for any suboperations op_1 and op_2 of q_1 and q_2, respectively, it holds that $op_1 \rightarrow op_2$.

A *system execution* σ over the register is a tuple $(\mathcal{A}, \rightarrow)$, where $\mathcal{A}$ is a set of operations on the register and $\rightarrow$ is the relation describing precedence among them, as above. Operations in $\mathcal{A}$ which are incomparable under $\rightarrow$ are called *overlapping*. It is assumed that there exists a write operation, which initializes the register, that precedes all other operations on it.

A *sequential execution* $(\mathcal{A}, \Rightarrow)$ over the register is one whose operations are *totally ordered* according to the transitive irreflexive order $\Rightarrow$. A sequential execution $(\mathcal{A}, \Rightarrow)$ satisfies the *sequential specification* of the register if each read operation in $\mathcal{A}$ returns the value written by the most recent write operation according to $\Rightarrow$.

A system execution $(\mathcal{A}, \rightarrow)$ is *atomic* or *linearizable* [7, 6] if it is equivalent to a sequential execution over the register which satisfies the sequential specification of the register.

A construction implements an *atomic* register if all its possible system executions are atomic.

For the proof of the atomicity of our construction we will use the following atomicity criterion for single-writer register constructions, which is due to Lamport [2]:

LEMMA 2.1. LAMPORT'S ATOMICITY CRITERION [2]: A register construction is atomic if there exists a *reading function* π, which, for any system execution $\sigma = (\mathcal{A}, \rightarrow)$: (i) assigns to each read $r \in \mathcal{A}$ a write $w \in \mathcal{A}$, such that the value returned by r —according to the read procedure invoked— is the value written by w and (ii) satisfies the following three conditions:

No-Future: For any read operation r of σ it is not the case that: $r \rightarrow \pi(r)$.

No-Past: For any read operation r of σ there is no write operation w such that $\pi(r) \rightarrow w \rightarrow r$.

No-New-Old-Inversion: For any two read operations r_1 and r_2 of σ it is not the case that: $(r_1 \rightarrow r_2$ and $\pi(r_2) \rightarrow \pi(r_1))$.

Roughly speaking, the above conditions imply that each read returns a written but not overwritten value and that reads obtain correctly-ordered values. (Actually the first two conditions alone are sufficient for regularity.)

Finally, the *cost measures* for the computation of space and time complexities of a construction $\mathcal{C}$ are:

- the number of subregisters used by $\mathcal{C}$ and
- the maximum number of suboperations on the subregisters performed during any read and any write operation in any system execution of $\mathcal{C}$.

3. DESCRIPTION OF THE CONSTRUCTION

The protocol is given in figure 2. We adopted the convention that shared variables are denoted by upper-case letters and local variables are denoted by lower-case letters. From the notation point of view we adopt the use of $\overline{x}$ to denote $1 - x$, where $x \in \{0, 1\}$. Also , $bin(i)$ denotes the binary representation of i in $\log k$ bits, $\oplus$ and $\circ$ represent bitwise exclusive-or and multiplication, respectively ($bin(i)$ multiplied by 0 is the zero-vector of length $\log k$ and $bin(i)$ multiplied by 1 is $bin(i)$ itself). Private variables are persistent, i.e. they retain their values between different invocations. The initializing conditions will be made clear after the presentation of the construction given in the following paragraphs.

The construction uses three sets of binary atomic registers, namely $H = \{H_1, \ldots, H_{k-1}\}$, $L^0 = \{L^0_1, \ldots, L^0_{k-1}\}$ and $L^1 = \{L^1_1, \ldots, L^1_{k-1}\}$. These subregisters are written by the writer and are read by the reader. One more binary register, RM, is used in order to allow the reader to pass a bit of information to the writer.

Assume that k —the number of values that the register under implementation will hold— is a power of 2. Later we will show how to

```
/* Shared variables declaration*/
var H_1,...,H_{k-1}, L^0_1,...,L^0_{k-1}, L^1_1,...,L^1_{k-1} ∈ {0,1} ;
                                   /* init. 0; only L^0_{v_0} = 1 */
    RM ∈ {0,1} ;
                                   /* Reader's Mode: init. 1 */

function COLOR(x_1,...,x_{2(k-1)})
                 /* COLOR : {0,1}^{2(k-1)} → {0,...,k-1} */
begin return(⊕_{i=1,...,k-1}((x_i ⊕ x_{k-1+i}) ∘ bin(i))) end

procedure READ
                          /* returns a value ∈ {0,...,k-1} */
var h_1,...,h_{k-1}, l^0_1,...,l^0_{k-1}, l^1_1,...,l^1_{k-1} ∈ {0,1} ;
                                                   /* init. 0 */
    rm, wm ∈ {0,1} ;                               /* init. 1 */
begin
    for i = 1 to k-1 do read H_i into h_i od ;
    wm := ⊕_{i=1,...,k-1} h_i ;
    if rm ≠ wm then
                          /* writer "moved" since last READ */
       for i = 1 to k-1 do read L_i^{¬wm} into l_i^{¬wm} od ;
       rm := wm ;
    endif
    write wm to RM ;
    for i = 1 to k-1 do read L_i^{rm} into l_i^{rm} od ;
    return(COLOR(h_{k-1},...,h_1, l^0_{k-1} ⊕ l^1_{k-1},..., l^0_1 ⊕ l^1_1)) ;
end

procedure WRITE(v)
                          /* writes value v ∈ {0,...,k-1} */
var h_1,...,h_{k-1}, l^0_1,...,l^0_{k-1}, l^1_1,...,l^1_{k-1} ∈ {0,1} ;
                                   /* init. same as shared var's */
    wm, rm ∈ {0,1} ;                               /* init. 0 */
    old, i ∈ {0,...,k-1} ;                         /* init. v_0 */
begin
    if v = old then exit ;
    compute i s.t. bin(i) = bin(v) ⊕ bin(old) ;
    read RM into rm ;
                                   /* check if reader "followed" */
    if rm = wm then
      wm := ¬wm ; h_i := ¬h_i ;
      write h_i to H_i ;
    else l_i^{wm} := ¬l_i^{wm} ; write l_i^{wm} to L_i^{wm} ;
    endif
    old := v ;
end
```

FIGURE 2. The atomic register protocol in pseudocode

remove this restriction. The $2(k-1)$-tuple: $(H_{k-1},\ldots,H_1,L^0_{k-1}\oplus L^1_{k-1},\ldots,L^0_1\oplus L^1_1)$ (in that order: High Order Bits, Low Order Bits) corresponds to a vertex of a $2(k-1)$-dimensional hypercube (recalling its definition: a *n-dimensional hypercube* is a regular undirected graph with 2^n vertices labelled from 0 to 2^n-1, where two vertices are connected if the binary representation of their labels differ in exactly one bit; figure 1 shows a 3-dimensional hypercube), which is colored using a function COLOR, which maps each vertex label ($2(k-1)$ bit string) to one of k colors ($\log k$ bit string), in a way such that each vertex has *exactly two* neighbors with each of the $k-1$ colors other than its own (*rainbow coloring*). Colors are in one-to-one correspondence with the value of the register. Thus, function COLOR can be used to extract the value of the register from the values of the subregisters of H, L^0 and L^1. An example of the coloring function COLOR for a 6-dimensional hypercube is given in figure 3. For reasons of readability of the figure, the connections that correspond to the high order bits of the vertex labels are not shown in full; instead three "representative" ones are drawn.

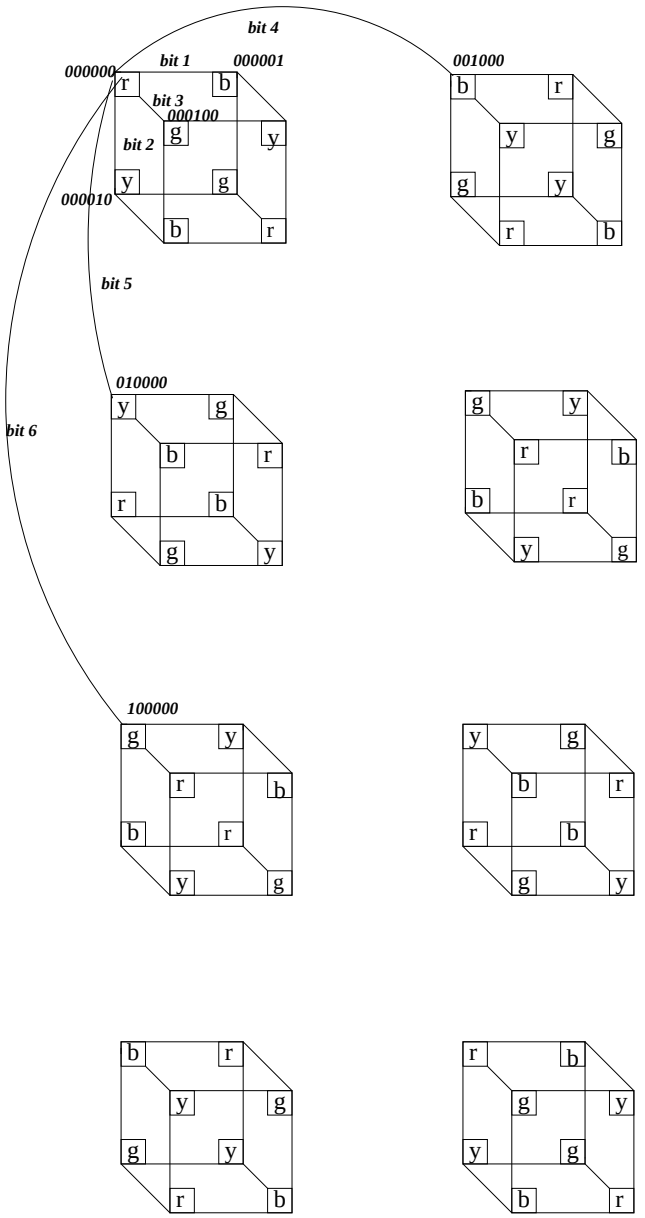

FIGURE 3. A 6-dimensional hypercube, rainbow 4-colored (r(ed), b(lue), y(ellow), g(reen)) using function COLOR

To ensure atomicity, the construction employs *hand-*

shaking. This mechanism implies that there are two "virtual places", also called *modes*, where the reader and the writer may be during each access to the register; the reader tries to be at the same place with the writer, while the latter tries to avoid it, by "moving" to the other virtual place when it sees that it has been "followed". By having disjoint sets of subregisters that can be accessed in each virtual place, the handshaking mechanism guarantees the existence of a piece of information that can be accessed by each communicating part without collision on the physical level.

The controller of the game here is the writer; in each write operation it has to:

1. determine the reader's mode by reading the subregister RM and
2. assign the new value to the register and change its virtual place place (i.e. change mode) if the reader has "followed".

From the particular rainbow property of the coloring function, it follows that the writer has the capability of changing the value of the register by modifying a single one of the construction's subregisters; moreover, in order to do so, it has two options: to modify either one of the High Order Bits (in H) or one of the Low Order Bits (in L^0 or L^1). E.g. in figure 3, if the binary registers values correspond to node 000000 in the hypercube (i.e. to value r(ed)) and the writer must write b(lue), it may do so by modifying either bit 4 (thus leading to node 001000) or bit 1 (thus leading to node 000001). The first option is taken when the writer has to change mode besides having to modify the register's value. (Therefore, a parity function of the subregister values of H can be used by the reader to trace the writer's mode each time.) The second option is taken when the writer needs only to change the register's value. Depending on which mode it is, it modifies one of the subregisters of either the set L^0 or the set L^1.

On the other hand, the reader, in each read operation, first assigns to its local variable wm the virtual place (mode) in which the writer is. It determines this from the values of the subregisters of H, using a parity function, as explained in the previous paragraph. If the writer has "moved" (changed mode) since the previous read, the reader reads the subregisters in $L^{\overline{wm}}$, in order to find which was the last configuration of that set when the writer had to "move" to virtual place wm. (Note that, if the writer has not "moved" since the previous read, the information in $L^{\overline{wm}}$ remains intact since it was last read.) After that, the reader updates RM in order to show to the writer that it has followed it in its new virtual place (mode). Subsequently, in either case, it reads the subregisters in L^{wm}. At that point the reader has a complete view of a recent enough (i.e. the most recent view that is not subject to be updated concurrently with it, i.e. after it started reading) set of values of H, L^0 and L^1 and it can use COLOR to extract the register's value from that information.

The construction components are *initialized* so that the reader's mode is 1 (RM is set to 1), the writer's mode is 0 and the register holds some initial value v_0 (all H_i and all L^1_i are set to 0; all L^0_i are set to 0, unless $v_0 \neq 0$, in which case $L^0_{v_0}$ is set to 1).

4. PROOF OF CORRECTNESS

First we prove that the encoding adopted using function COLOR is correct:

LEMMA 4.1. The function COLOR as defined in Figure 2 has the property that for all $x \in \{0,1\}^{2(k-1)}$ and for all $v \in \{0,\ldots,k-1\}$ if $v \neq \texttt{COLOR}(x)$ then there exist exactly two y_1 and y_2 which are both in $\{0,1\}^{2(k-1)}$ such that $v = \texttt{COLOR}(y_1) = \texttt{COLOR}(y_2)$, $y_1 \neq y_2$ and both y_1 and y_2 differ from x in exactly one bit.

Proof. It holds that for all $x, y \in \{0,1\}^{2(k-1)}$ such that x, y differ only in one bit (assume w.l.o.g either bit i or bit $k-1+i$, where $1 \leq i \leq k-1$), $\texttt{COLOR}(x) \neq \texttt{COLOR}(y)$ because $\texttt{COLOR}(x) \oplus \texttt{COLOR}(y) = bin(i)$ which is not zero.

Moreover, given x and y as above there exists another $y' \in \{0,1\}^{2(k-1)}$ such that $y' \neq y$ and y' differs from x only in one bit, either bit $k-1+i$ or bit i, respectively, and $\texttt{COLOR}(y) = \texttt{COLOR}(y')$. This is because $\texttt{COLOR}(y) \oplus \texttt{COLOR}(y') = 0$.

On the other hand, given again x and y as above, for any other $y'' \in \{0,1\}^{2(k-1)}$ which differs from x in only one bit excluding bits i and $k-1+i$ (x and y'' differ either in bit j or in bit $k-1+j$, $1 \leq j \leq k-1$ and $j \neq i$) it holds that $\texttt{COLOR}(y) \neq \texttt{COLOR}(y'')$ because $\texttt{COLOR}(y) \oplus \texttt{COLOR}(y'') = bin(i) \oplus bin(j)$ which is not zero, because $i \neq j$. □

Next we focus in proving the atomicity of our construction. We introduce some auxiliary terminology, which helps the presentation:

- For a read operation r, $put(r)$ denotes its subwrite to RM, $mode(r)$ is the value it writes to RM, while $view(r)$ is the $3(k-1)$-tuple of values (h, l^0, l^1) that it uses in its invocation of COLOR. We write $view(r)|H$ (resp. $view(r)|L^0$, $view(r)|L^1$), for the restriction of the view to the tuple corresponding to the values of h (resp. l^0, l^1).
  For a write operation w, $get(w)$ denotes its subread from RM, $mode(w)$ is the value of the writer's local variable wm when w performs its (unique) subwrite operation, while $view(w)$ is the tuple consisting of the values of H, L^0, L^1 after the unique subwrite. Observe that $mode(w)$ equals $\oplus_{i=1,\ldots,k-1} H_i$ then. Here, too, we write $view(w)|H$ (resp. $view(w)|L^0$, $view(w)|L^1$), for the restriction of the view to the tuple corresponding to the values of H (resp. L^0, L^1).
- For a read operation r, let each one of its read suboperations be mapped to the most recent write operation which modified the respective subregister

(according to the total order defined on the operations of the atomic subregister). We define $\rho(r)$ to be the write operation of this set such that every other operation of this set precedes it. (If the only operation in this set is the initial write operation w_{init}, which, must naturally write (initialize) all the subregisters, then $\rho(r) = w_{init}$.) This is a well defined function because the write operations are totally ordered, since there are no overlapping writes.

As already indicated earlier in the paper, we will use Lamport's atomicity criterion (lemma 2.1) to prove the atomicity of our construction. Consider any arbitrary system execution σ. We first prove a couple of auxiliary lemmas (lemmas 4.2 and 4.3). These are followd by three lemmas on important properties of the handshaking implementation (lemmas 4.4, 4.5 and 4.6), namely that (i) while a read r scans the subregisters in H (resp. $L^{mode(r)}$), there can be at most one write that may concurrently attempt a modification in the same set; and that (ii) if a read r scans $L^{\overline{mode(r)}}$, there is no write that may concurrently a attempt to modify a register in that set (i.e. that scan is "collision-free"), while if it does not scan $L^{\overline{mode(r)}}$, then it does not miss any information, since the values in $L^{\overline{mode(r)}}$ could not have been modified since the last time that they were read. These properties are then used to prove (lemma 4.7) that ρ can play the role of the reading function π in Lamport's atomicity criterion. Then we show that the construction satisfies the three conditions in the criterion ((lemma 4.9).

LEMMA 4.2. For any read r and for any write w, such that $put(r) \rightarrow get(w)$ and $(\neg\exists \text{ read } r' : put(r) \rightarrow put(r') \rightarrow get(w))$, it is $mode(r) = \overline{mode(w)}$.

Proof. Since there are no overlapping reads, the lemma hypothesis implies that r is the last read to modify RM before w reads it. Thus w will read from RM into its local variable rm the value that r wrote; either this value will be complementary to the value of w's local variable wm, or w will complement wm. In both cases, due to the definitions of $mode(r)$, $mode(w)$, the lemma follows. □

LEMMA 4.3. In any sequence of successive writes with the same mode, only the first one could write to a subregister of the set H.

Proof. From the protocol it follows that a write w has a different mode compared to its directly preceding write iff it writes to a subregister of the set H. □

LEMMA 4.4. For any read r there can be at most one write whose write suboperation occurs between the start of r (i.e. $s(r)$) and $put(r)$ and modifies a subregister in the set H.

Proof. Since there are no overlapping reads, we can use induction on the number of reads that occur in any arbitrary system execution σ.

Let r_i denote the ith read in the execution. For the induction basis, suppose, towards a contradiction, that there exist write operations

$$w_x \rightarrow w_{x+1} \rightarrow \ldots \rightarrow w_{x+q} \rightarrow put(r_1) \ (q \geq 1)$$

whose write suboperations modify subregisters in H and occur between the first suboperation of r_1 and $put(r_1)$. From the initialization conditions it follows that each write w such that $get(w) \rightarrow put(r_1)$ sees that $RM = 1$ and its local variable $wm = 0$; therefore, according to the writer's protocol, w will not write in H. This leads to the desired contradiction, which proves the induction basis.

For the induction hypothesis, assume that the lemma holds for all r_k, $1 \leq i \leq k$. We will show that it also holds for r_{k+1}. Assume, towards a contradiction that there exist more than one write operations

$$w_x \rightarrow w_{x+1} \rightarrow \ldots \rightarrow w_{x+q} \rightarrow put(r_{k+1}) \ (q \geq 1)$$

whose write suboperations modify subregisters in H and occur between $s(r_{k+1})$ and $put(r_{k+1})$. There are two cases to consider:

$put(r_k) \rightarrow get(w_x)$: Note that

$$put(r_k) \rightarrow get(w_x) \rightarrow \ldots \rightarrow get(w_{x+q}) \rightarrow put(r_{k+1})$$

By use of lemma 4.2 all $w_x, w_{x+1}, \ldots w_{x+q}$ have the same mode (i.e. $\overline{mode(r_k)}$). Note also that all the other possibly existing intermediate writes (between w_x and w_{x+q}) also have the same mode, since writes change mode only by writing to subregisters in H. But then, by lemma 4.3, only the first of the whole sequence of writes (i.e. w_x) may write on a subregister of H, hence we have a contradiction.

$get(w_x) \rightarrow put(r_k)$: From the induction hypothesis we know that between $s(r_k)$ and $put(r_k)$ only one subwrite to a subregister in H occurs. We can modify execution σ into one σ' where the subwrite of w_x occurs between $s(r_k)$ and $put(r_k)$, but at a time so that r_k "misses" it (i.e. r_k reads the subregister modified by w_x before the subwrite takes place). Execution σ' is equivalent to σ, as the only two operations involved in the modification behave in exactly the same way and do not affect the behaviour of any other operation; hence, every operation in σ' behaves the same as its equivalent in σ. By the induction hypothesis we know that between $s(r_k)$ and $put(r_k)$ only one subwrite to a subregister in H occurs. In this case it is the subwrite by w_x, which is "missed" by r_k, Hence, if $mode(w_x) = m$ then $mode(r_k) = \overline{m}$. Note also that

$$put(r_k) \rightarrow w_{x+1} \rightarrow \ldots \rightarrow w_{x+q} \rightarrow put(r_{k+1})$$

This implies, by lemma 4.2, that the writes $w_{k+1}, \ldots w_{k+q}$ also have mode equal to m. Note

also that all the other possibly existing intermediate writes (between w_x and w_{x+q}) also have the same mode, since writes change mode only by writing to subregisters in H. But then, by lemma 4.3, only the first of the whole sequence of writes (i.e. w_x)may write on a subregister of H, hence we have a contradiction, as in the previous case.

□

LEMMA 4.5. For any read r with $mode(r) = m$, there can be at most one write whose write suboperation occurs between $put(r)$ and the finishing time of r (i.e. $f(r)$) and modifies a subregister in the set L^m.

Proof. Assume, towards a contradiction that there exist more than one write operations

$$w_x \rightarrow w_{x+1} \rightarrow \ldots \rightarrow w_{x+q} \rightarrow f(r) \ (q \geq 1)$$

whose write suboperations modify subregisters in L^m and occur between $put(r)$ and $f(r)$. This implies that

$$put(r) \rightarrow get(w_{x+1}) \rightarrow \ldots \rightarrow get(w_{x+q}) \rightarrow f(r)$$

By lemma 4.2, then, it will be that $mode(w_{x+1}) = \ldots = mode(w_{x+q}) = \overline{m}$, hence their write suboperations will be on subregisters in H or in $L^{\overline{m}}$, which contradicts our assumption. □

LEMMA 4.6. For any read r with $mode(r) = m$,
1. if r scans the subregisters in $L^{\overline{m}}$, there is no write which modifies a subregister in $L^{\overline{m}}$ and whose write suboperation occurs during that scanning interval of r.
2. if r does not scan the subregisters in $L^{\overline{m}}$ and r^- is the read directly preceding r in the execution (r^- exists since the first read in any execution scans $L^{\overline{m}}$), there is no write which modifies a subregister in $L^{\overline{m}}$ and whose write suboperation occurs between $put(r^-)$ and $put(r)$.

Proof. *1.* Assume, towards a contradiction that there exists such a write w. Since r scans $L^{\overline{m}}$, either r is the first read in the execution or there exists a read r^- directly preceding r with $mode(r^-) = \overline{m}$.

In the former case we directly come to a contradiction, since, due to the initializing conditions, any write w such that $get(w) \rightarrow put(r)$ has $mode(w) = 0$ and writes to L^0, while $mode(r)$ is also 0.

In the latter case, it must be $get(w) \rightarrow put(r^-)$ (otherwise, by lemma 4.2 it would be $mode(w) = m$ and hence, w would not write in $L^{\overline{m}}$). Since r scans $L^{\overline{m}}$ after having scanned H and we assumed that w's subwrite in $L^{\overline{m}}$ occurs during the scan of $L^{\overline{m}}$ by r, this implies that during the interval that r scans the subregisters in H there is no write that modifies any of them and it holds that $\oplus_{i=1,\ldots,k-1} H_i = \overline{m}$. But then it should be that $mode(r) = \overline{m}$, hence we have a contradiction.

2. Assume, towards a contradiction that there exists such a write w. Since r does not scan $L^{\overline{m}}$, it must be $mode(r^-) = m$, as well. Then the following are true:

a. The assumptions combined imply that there must be a w^-, such that $w^- \rightarrow w$, with $mode(w^-) = \overline{m}$, which modifies a subregister in H and this modification is "missed" by r^- (otherwise it would be $mode(r^-) = \overline{m}$), but read by r.

b. Any other write w' that possibly modifies a subregister in H and $put(r^-) \rightarrow w' \rightarrow put(r)$ (i.e. is possibly "seen" by r) has $mode(w') = \overline{m}$.

These combined imply that $mode(r) = \overline{m}$, hence we have a contradiction.

□

The following is the main lemma to use in order to apply Lamport's atomicity criterion.

LEMMA 4.7. For any read r, $view(r) = view(\rho(r))$.

Proof. Let m be the value of $mode(r)$.

If $\rho(r) \rightarrow r$, then the lemma clearly holds.

If $\rho(r)$ is overlapping r and its subwrite is on a subregister in H, then, by lemma 4.4, $view(r)|H = view(\rho(r))|H$. Also, $view(r)|L^m = view(\rho(r))|L^m$, otherwise the definition of $\rho(r)$ would be contradicted, since r scans the subregisters in L^m after having scanned the subregisters in H. The same holds for $view(r)|L^{\overline{m}}$ and $view(\rho(r))|L^{\overline{m}}$ if r scans $L^{\overline{m}}$. If r does not scan $L^{\overline{m}}$, by lemma 4.6 we have that $L^{\overline{m}}$ was not modified since the last scan, hence $view(r)|L^{\overline{m}} = view(\rho(r))|L^{\overline{m}}$ in this case also.

If $\rho(r)$ is overlapping r and its subwrite is on a subregister in L^m, then, by lemma 4.5, $view(r)|L^m = view(\rho(r))|L^m$. Also, the fact that $mode(r) = m$ implies that the last modification to H (which precedes $\rho(r)$ and either precedes or overlaps r as in lemma 4.4) was "seen" by r (otherwise its mode would have been equal to $\overline{m}$), hence $view(r)|H = view(\rho(r))|H$. Lemma 4.6 directly implies that $view(r)|L^{\overline{m}} = view(\rho(r))|L^{\overline{m}}$, as well.

Note that it cannot be the case that $\rho(r)$ is overlapping r and its subwrite is on a subregister in $L^{\overline{m}}$. If this was the case, then, due to the first part of lemma 4.6, the subwrite of $\rho(r)$ in $L^{\overline{m}}$ would have happened before r started to scan $L^{\overline{m}}$. But the fact that $mode(r) = m$ would imply that there exists a write w with $mode(w) = m$, modifying H, such that $\rho(r) \rightarrow w$, and r "sees" the modification of w. This would contradict the definition of $\rho(r)$. □

From the previous lemma it follows that:

COROLLARY 4.8. For any read r, $mode(r) = mode(\rho(r))$.

LEMMA 4.9. The protocol satisfies lemma 2.1 (Lamport's Atomicity Criterion).

Proof. We use $\rho(r)$ as $\pi(r)$, which, by lemma 4.7 assigns to each read r a write w, such that the value returned by r —according to the read procedure invoked— is the value written by w. We next prove the three conditions that are sufficient to imply atomicity.

No-Future: From the definition of $\rho(r)$ it follows that the last suboperation of $\rho(r)$ occurs before the last suboperation of r.

No-Past: Suppose, towards a contradiction, that there exist a read r and a write w of σ such that $\rho(r) \rightarrow w \rightarrow r$. Let $mode(r) = m$. If w writes in H or L^m, we have directly a contradiction to the definition of $\rho(r)$, since a read with mode m always scans H and L^m. Hence, the only possibility is that w writes in $L^{\overline{m}}$. Note that this implies that $mode(w) = \overline{m}$. Since, due to corollary 4.8 $mode(\rho(r)) = m$ and, due to the protocol, a write has different mode compared to its directly preceding write iff it writes to a subregister in H, we have that there must be a write w' that writes to a subregister in H, and $\rho(r) \rightarrow w' \rightarrow w \rightarrow r$. Hence, either $mode(r) = \overline{m}$ (a contradiction to the assumption) or there exists another write w'' such that $\rho(r) \rightarrow w' \rightarrow w \rightarrow w''$ that modifies H and is "seen" by r (a contradiction to the definition of $\rho(r)$.

No-New-Old-Inversion: Suppose, towards a contradiction that $\exists$ reads r_1, r_2 in σ such that $r_1 \rightarrow r_2$ and $\rho(r_2) \rightarrow \rho(r_1)$. From the definition of $\rho(r_1)$ it follows that the last suboperation of $\rho(r_1)$ occurs before the last suboperation of r_1. This implies that $\rho(r_2) \rightarrow \rho(r_1) \rightarrow r_2$, since $r_1 \rightarrow r_2$. But this is a contradiction to the *No-Past* condition, which has already been shown to hold.

□

For the case that k is not power of 2, the protocol can use $3l - 2$ subregisters, where $l = 2^{\lceil \log k \rceil}$, i.e. l is the smallest power of 2 larger than k. In this way the protocol will in fact implement an l-valued atomic register ($k < l$), which can also serve as a k-valued one. Hence, we have the following:

THEOREM 4.10. The presented construction correctly implements a wait-free k-valued atomic register using $3 \cdot 2^{\lceil \log k \rceil} - 2$ atomic binary subregisters. The maximum number of suboperations performed during any read r is $3 \cdot 2^{\lceil \log k \rceil} - 3$, while each write w performs one read and one write suboperation.

ACKNOWLEDGEMENTS

This work was done while the authors were guest researchers at CWI, Amsterdam and were partially supported by the ESPRIT II BRA Program of the European Community, under contract no. 7141 (project ALCOM II). The first author was also partially supported by a NUFFIC Fellowship. The second author was also partially supported by NWO through NFI Project ALADDIN under contract number NF 62-376.

The authors gratefully acknowledge Jaap-Henk Hoepman, John Tromp and Paul Vitányi for all the helpful discussions.

Special thanks also goes to the anonymous referees, whose constructive comments helped to improve the readability of the paper.

REFERENCES

[1] P.J. Courtois, F. Heymans and D.L. Parnas. (1971) Concurrent Control With Readers and Writers. *Communication of the ACM*, **14**(10), 667-668.

[2] L. Lamport. (1986) On Interprocess Communication, Part I: Basic Formalism, Part II: Basic Algorithms. *Distributed Computing*, **1**, 77-101.

[3] N. Lynch (1996) *Distributed Algorithms*. Morgan Kaufmann.

[4] B. Bloom. (1988) Constructing Two-writer Atomic Registers. *IEEE Transactions on Computers*, **37**, 1506–1514.

[5] J.E. Burns and G.L. Peterson. (1987) Constructing Multi-reader Atomic Values From Nonatomic Values. *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, 222–231.

[6] M. P. Herlihy. "Wait-free synchronization". *ACM Transactions on Programming Languages and Systems* **13**, 1 (1991), 124–149.

[7] M. Herlihy and J. Wing. (1990) Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming, Languages and Systems* **12**(3), 463–492.

[8] A. Israeli and A Shaham. (1992) Optimal Multi-Writer Multi-reader Atomic Registers. *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, 71–82.

[9] L.M. Kirousis, E.Kranakis, P.M.B. Vitányi. (1987) Atomic Multireader Register. *Proceedings of the 2nd International Workshop on Distributed Algorithms*, vol. 312 of LNCS, Springer-Verlag, 278–296.

[10] M. Li, J.Tromp and P.M.B. Vitányi. (1986) How to Construct Concurrent Wait-free Variables. *Journal of the ACM*, **43**(4), 723–746.

[11] R. Newman-Wolfe. (1987) A Protocol for Wait-free, Atomic, Multi-reader Shared Variables. *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, 232–248.

[12] G.L. Peterson and J.E. Burns. (1987) Concurrent Reading While Writing II: The Multiwriter Case. *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, 383–392.

[13] G.L. Peterson. (1983) Concurrent Reading While Writing. *ACM Transactions on Programming Languages and Systems* **5**(1), 46–55.

[14] R. Schaffer. (1988) On the Correctness of Atomic Multi-writer Registers. Technical Report *MIT/LCS/TM-364*, MIT Lab. for Computer Science.

[15] A.K. Singh, J.H. Anderson and M.G. Gouda. (1987) The Elusive Atomic Register Revisited. *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, 206–221.

[16] J. Tromp. (1989) How to Construct an Atomic Variable. *Proceedings of the 3rd International Workshop on Distributed Algorithms*, vol. 392 of LNCS, Springer-Verlag, 492–302.

[17] K. Vidyasankar. (1988) "Converting Lamport's Regular Register to Atomic Register". *Information Processing Letters*, **28**, 287–290.

[18] K. Vidyasankar. (1989) An Elegant 1-Writer Multireader Multivalued Atomic Register. *Information Processing Letters*, **30**, 221–223.

[19] K. Vidyasankar. (1990) Concurrent Reading While Writing Revisited. *Distributed Computing*, **4**, 81–85.

[20] P. Vitányi and B. Awerbuch. (1986) Atomic Shared Register Access by Asynchronous Hardware. *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, 233–243.

[21] S. Chaudhuri, M.J. Kosa and J.L. Welch. (1991) Upper and Lower Bounds for One-Write Multivalued Regular Registers. *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, 134–141.

[22] S. Chaudhuri and J.L. Welch. (1990) Bounds on the Costs of Register Implementations. *Proceedings of the 4th International Workshop on Distributed Algorithms*, vol. 486 of LNCS, Springer-Verlag, 402–421.

[23] P. Jayanti, A. Sethi and E.L. Lloyd. (1992) "Minimal Shared Information for Concurrent Reading and Writing". *Proceedings of the 5th International Workshop on Distributed Algorithms*, vol. 579 of LNCS, Springer-Verlag, 212–228.

[24] G. Kant and J. van Leeuwen. (1990) The File Distribution Problem for Processor Networks. *Proceedings of the Second Scandinavian Workshop on Algorithm Theory*, vol. 447 of LNCS, Springer-Verlag, 48–59.

[25] L.M. Kirousis, P. Spirakis, Ph. Tsigas. (1994) Reading Many Variables in One Atomic Operation: Solutions With Linear or Sublinear Complexity. *IEEE Transactions on Parallel and Distributed Systems*, **5**(7), 688-696. (Prel. version in *Proceedings of the 5th International Workshop on Distributed Algorithms*, vol. 579 of LNCS, Springer-Verlag, 229–241.)

[26] C. Dwork, M. Herlihy, S. Plotkin and O. Waarts. (1992) Time-Lapse Snapshots. *Proceedings of the First Israel Symposium on the Theory of Computing and Systems*, vol. 601 of LNCS Springer-Verlag, 154–170.

[27] C. Dwork, O. Waarts. (1992) Simple and Efficient Bounded Concurrent Timestamping or Bounded Concurrent Timestamp Systems are Comprehensible! *Proceedings of the 24th ACM Symposium on Theory of Computing*, 656–666.