How a Rainbow Coloring Function Can Simulate Wait-free Handshaking

M. Papatriantafilou

Computer Science/Department of Algorithmics and Architecture

CS-R9437 1994

How a Rainbow Coloring Function Can Simulate Wait-Free Handshaking

Marina Papatriantafilou
Email: ptrianta@cwi.nl

Philippas Tsigas
Email:tsigas@cwi.nl

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
& Computer Technology Institute, Patras, Greece
& Computer Science Dept., University of Patras, 26500 Patras, Greece

Abstract

How to construct shared data objects is a fundamental issue in asynchronous concurrent systems, since these objects provide the means for communication and synchronization between processes in these systems. Constructions which guarantee that concurrent access to the shared object by processes is free from waiting are of particular interest, since they may help to increase the amount of parallelism in such systems. 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*) has received some attention, 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 which uses k binary safe registers and exploits the properties of a rainbow coloring function of a hypercube. The best known construction for the regular/atomic case uses $\binom{k}{2}$ binary regular/atomic registers. In this work we show how the rainbow coloring function can be extended to simulate a handshaking mechanism between the reader and the writer of the register, thus offering a solution for the atomic register case with *one* reader, which uses only $3k-2$ binary registers. The lower bound for such a construction is $k-1$.

AMS Subject Classification (1991): 94A99, 68Q10, 68Q22, 68Q25
CR Subject Classification (1991): D.4.5, E.4, F.1.2, G.2.2
Keywords & Phrases: Asynchrony, Distributed Computing, Fault-tolerance, Graph Coloring, Hypercube, Shared Memory Systems, Wait-Free Constructions
Note: 1. This work was partially supported by the ESPRIT II Basic Research Actions Program of the EC 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.
Note: 2. This paper will appear in the Proceedings of the *19th International Symposium on Mathematical Foundations of Computer Science* (*MFCS '94*).

1. Introduction

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 them is to require that a write have exclusive access to the data, thus making only concurrent reading possible [5]. 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. However, 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 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 by a number of processes in a wait-free manner is also called a *wait-free shared register*; from now on we adopt the convention to call it *register*. Registers can be classified according to the strength of 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 [14] and have become of fundamental importance in the study of shared registers. According to those definitions: (i) A register is called *safe* if it guarantees only that a read operation 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; this value may equal any possible value of the register. (ii) A register is called *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. (iii) A register is called *atomic* if, besides ensuring safeness, it guarantees that although read and write operations may 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. Except from the above classification, registers are also distinguished by the number of readers that may concurrently read the register, the number of writers that may concurrently write the register, as well as the number of values it can take on. All these dimensions imply a hierarchy on registers, with single-reader, single-writer boolean safe register in the lowest level and multireader, multiwriter, multivalued atomic variables in the highest level.

Despite the fact that there has been a great deal of research on developing implementations of stronger registers out of weaker ones [1, 2, 9, 10, 12, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25], to the best of our knowledge, comparatively few results have appeared studying the costs incurred by such implementations [3, 4, 11, 21]. Chaudhuri and Welch in [4] 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 is a problem with multiple cases to be examined. As a second step 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 considered are the number of subregisters used in the register construction and the number of subreads and subwrites performed during each read or write operation of the registers, respectively. 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 rainbow coloring function of the $(k-1)$-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 [8] independently have shown the same result for the rainbow coloring of a hypercube and they applied it to the file distribution problem. A *rainbow coloring* of a hypercube with k colors is a coloring such that each node of the hypercube has a neighbor with each one of the $k-1$ colors other than its own. In a subsequent paper [3], Chaudhuri, Kosa and Welch give a construction of a k-valued regular/atomic register out of binary regular/atomic ones in which each write performs only one subwrite to one of them. This algorithm requires $\binom{k}{2}$ subregisters. It must be pointed out that both these two algorithms in [4] and in [3] are for the multi-reader case while they have the writer perform *only one* suboperation per write, the one subwrite.

In this paper we show how a rainbow coloring —which has been used in order to construct the weakest of the k-valued from 2-valued registers, namely the safe one— can be extended to simulate a powerful handshaking mechanism (see Tromp [21], Spirakis, Kirousis, Tsigas [13] and Dwork et al. [6, 7]). Using this simulation we can 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 lower bound for such a construction is $k-1$. To the best of our knowledge this is the first linear one-write construction.

2. Model of a Register Construction

A *shared register* is an abstract data object shared by a number of concurrent processes which may either read or write it. A *construction* of a register comprises of i) a data structure consisting of memory cells called *subregisters*, ii) a set of initial values for the subregisters and iii) a set of read and write procedures which provide the means to the processes to access the register; these procedures are also referred as *protocol*. 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 *operation 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. In order 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 any operation will complete in a finite number of steps. Roughly speaking, the wait-free condition rules out unbounded busy waiting as well as conditional waiting. The reason for the former is obvious, while the latter holds because otherwise a process might be executing an infinite number of steps waiting for a condition to become true by a crashed process.

In a global time model each operation q is assumed to have a time interval $[s_q, f_q]$ on one linear time axis ($s_q < f_q$). Think of s_q and f_q as the starting and finishing time instants of q. During this time interval the operation is said to be *pending.* There is a precedence relation on operations (denoted by '$\rightarrow$'), which is a strict partial order. $q_1 \rightarrow q_2$ means that operation q_1 ends before operation q_2 starts. If two operations are incomparable under $\rightarrow$, they are said to *overlap.* 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 *reading function* π for a register construction $\mathcal{C}$ is a function that assigns a write operation w to each read operation r on the register, such that the value returned by r —according to the read procedure invoked— is the value written by w. It is assumed that there exists a write operation, which initializes the register, that precedes all other operations on it.

Let $\mathcal{A}$ denote the set of read and write procedures of the construction. A triple $\sigma = (\mathcal{A}, \rightarrow, \pi)$ is called a *system execution* of $\mathcal{C}$.

A construction is said to implement a *regular* register if all its system executions are regular. A system execution $\sigma = (\mathcal{A}, \rightarrow, \pi)$ is *regular* if for any read operation r of σ i) not $r \rightarrow \pi(r)$ and ii) there is no write w such that $\pi(r) \rightarrow w \rightarrow r$. A construction is said to implement an *atomic* register if all its system executions are atomic. A system execution $\sigma = (\mathcal{A}, \rightarrow, \pi)$ is *atomic* if $\rightarrow$ can be extended to a total order $\Rightarrow$ such that for any read operation r of σ i) $\pi(r) \Rightarrow r$ and ii) there is no write w such that $\pi(r) \Rightarrow w \Rightarrow r$. From the respective definitions it can be noticed that an atomic construction is also regular.

Finally, the *cost measures* for the computation of space and time complexities of a construction $\mathcal{C}$ are: i) the number of subregisters used by $\mathcal{C}$ and ii) 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 construction presented here 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 additional binary register RM is used in order to allow the reader to pass a bit of information to the writer, for the sake of the handshaking mechanism.

We adopt the convention that shared variables are denoted by upper-case and local variables are denoted by lower-case. From the notation point of view we adopt the use of $\overline{x}$ to denote $1-x$, where $x \in \{0, 1\}$. From now on, 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 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 with a vertex of a $2(k-1)$-dimensional hypercube (recalling its definition: a $2(k-1)$-*dimensional hypercube* is a regular undirected graph with $2^{2(k-1)}$ vertices labeled from 0 to $2^{2(k-1)} - 1$, where two vertices are connected if their labels differ in exactly one bit). This hypercube is colored using a function f, 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

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

function $f(x_1,\dots,x_{2(k-1)})$                  /* $f : \{0,1\}^{2(k-1)} \to \{0,\dots,k-1\}$ */
begin return($\oplus_{i=1,\dots,k-1}((x_i \oplus x_{k-1+i}) \circ bin(i))$) end

procedure READ                                      /* returns a value $\in \{0,\dots,k-1\}$ */
var $h_1,\dots,h_{k-1}, l^0_1,\dots,l^0_{k-1}, l^1_1,\dots,l^1_{k-1} \in \{0,1\}$ ;   /* init. 0 */
    $rm, wm \in \{0,1\}$ ;                          /* init. 1 */
begin
    for $i = 1$ to $k-1$ do read $H_i$ into $h_i$ od ;
    $wm := \oplus_{i=1,\dots,k-1} h_i$ ;
    if $rm \neq wm$ then                            /* writer "moved" since last READ */
        for $i = 1$ to $k-1$ do read $L^{\overline{wm}}_i$ into $l^{\overline{wm}}_i$ od ;
        $rm := wm$ ;
    endif
    write $wm$ to $RM$ ;
    for $i = 1$ to $k-1$ do read $L^{rm}_i$ into $l^{rm}_i$ od ;
    return($f(h_{k-1},\dots,h_1, l^0_{k-1} \oplus l^1_{k-1},\dots,l^0_1 \oplus l^1_1)$) ;
end
procedure WRITE($v$)                                /* writes value $v \in \{0,\dots,k-1\}$ */
var $h_1,\dots,h_{k-1}, l^0_1,\dots,l^0_{k-1}, l^1_1,\dots,l^1_{k-1} \in \{0,1\}$ ;  /* init. same as shared var's */
    $wm, rm \in \{0,1\}$ ;                          /* init. 0 */
    $old, i \in \{0,\dots,k-1\}$ ;                  /* init. $v_0$ */
begin
    if $v = old$ then exit ;
    compute $i : bin(i) := bin(v) \oplus bin(old)$ ;
    read $RM$ into $rm$ ;                           /* check if reader "followed" */
    if $rm = wm$ then $wm := \overline{wm}$ ; $h_i := \overline{h_i}$ ; write $h_i$ to $H_i$ ;
    else $l^{wm}_i := \overline{l^{wm}_i}$ ; write $l^{wm}_i$ to $L^{wm}_i$ ;
    endif
    $old := v$ ;
end
```

Figure 1: The protocol

has *exactly two* neighbours with each one of the $k-1$ colors other than its own (*rainbow coloring*). Each color is in one-to-one correspondence with one value of the register under implementation. Thus, f can be used as a function that extracts the value of the register from the values of the subregisters of H, L^0 and L^1.

In order to ensure atomicity, the construction employs handshaking. 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 is the writer, who, in each write operation, has to: 1) determine the reader's mode by reading the subregister RM and 2) assign the new value to the register and change place if it has been followed by the reader. From the particular rainbow property of the coloring function as described above, 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). The first option is taken when the writer has to change mode besides having to modify the register's value. (This implies that a parity function of the subregister values of H can be used by the reader in order 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 mode in which the writer is. This can be determined from the values of the subregisters of H, using a parity function, as was 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. (Notice 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. Subsequently, in any case, it reads the subregisters in L^{wm}. At that point the reader has a complete view of a recent enough set of values of H, L^0 and L^1 and it can use f 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 its initial value v_0 (all H_i and all L_i^1 are set to 0; all L_i^0 are set to 0, unless $v_0 \neq 0$, in which case $L_{v_0}^0$ is set to 1).

The protocol is formally described in figure 1. There, $bin(i)$ denotes the binary representation of i in $\log k$ bits, $\oplus$ represents exclusive-or and $\circ$ represents multiplication ($bin(i)$ multiplied by bit 0 is the zero-vector of length $\log k$ and $bin(i)$ multiplied by bit 1 is $bin(i)$ itself). An example of the coloring function f for a 6-dimensional hypercube is given in figure 2. For reasons of readability of the figure, the connections that correspond to the high

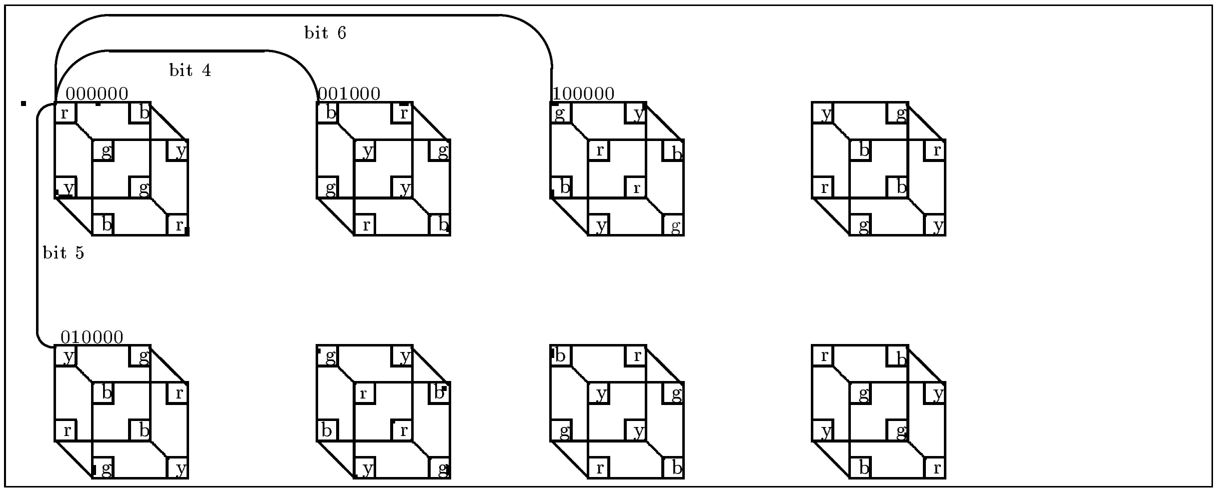

Figure 2: A 6-dimensional hypercube rainbow 4-colored using f

order bits of the vertex labels are not shown in full; instead three "representative" ones are drawn.

4. Correctness Proof of the Construction

First we prove that the encoding adopted using f is correct:

Lemma 1 *The function f as defined in figure 1 has the property that for all $x \in \{0,1\}^{2(k-1)}$ and for all $v \in \{0,\dots,k-1\}$ if $v \neq f(x)$ then there exist y_1 and y_2 which are both in $\{0,1\}^{2(k-1)}$ such that $v = f(y_1) = f(y_2)$, $y_1 \neq y_2$ 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 $1 \leq i \leq k-1$), $f(x) \neq f(y)$ because $f(x) \oplus f(y) = bin(i)$ 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 $f(y) = f(y')$. This is because $f(y) \oplus f(y') = 0$. 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 $f(y) \neq f(y'')$ because $f(y) \oplus f(y'') = bin(i) \oplus bin(j)$ which is not zero, because $i \neq j$. $\square$

From now on we concentrate in proving the atomicity of our construction. First we introduce some *auxiliary terminology*, which will help the presentation of our arguments:

- 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 $2(k-1)$-tuple of values that it uses as input in its invocation of f. For a write operation w, $get(w)$ denotes its subread from RM, $mode(w)$ is the value of the writer's local variable wm immediately before w performs its (unique) subwrite operation, while $view(w)$ is the $2(k-1)$-tuple consisting of the values : $(H_{k-1}, \ldots, H_1, L^0_{k-1} \oplus L^1_{k-1}, \ldots, L^0_1 \oplus L^1_1)$ immediately after the subwrite.

- A *phase of writes* $\mathcal{W}$ is a sequence of write operations $w_1, \ldots, w_n$ such that $w_1 \rightarrow \ldots \rightarrow w_n$ and $mode(w_1) = \ldots = mode(w_n) = m$ and for which there exist w_0 (if w_1 is not the first write operation of the respective execution σ) and w_{n+1} such that w_0 directly precedes w_1, w_{n+1} is directly preceded by w_n and $mode(w_0) = mode(w_{n+1}) = \overline{m}$.

- 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 actions 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. This function is well defined because the write operations are totally ordered, since there are no overlapping writes. A read operation r is called *related* to a phase of writes $\mathcal{W}$, if $\rho(r)$ is one of the writes in $\mathcal{W}$.

Lemma 2 *For any read r and for any write w, such that $put(r) \rightarrow get(w)$ and $(\neg\exists$ 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 3 *Let $\mathcal{W}$ be the phase of writes related to a read operation r. Then there is no write operation w in $\mathcal{W}$ such that $put(r) \rightarrow get(w)$.*

Proof. Since the unique subwrite operation of any write operation is also its last suboperation, $\rho(r)$ either precedes or overlaps r. Let $m = mode(r)$. Lemma 2 implies that each write w, which overlaps r and $put(r) \rightarrow get(w)$, writes in one of the subregisters in $L^{\overline{m}}$ or H. But after $put(r)$, r will read L^m. Thus, it cannot be $put(r) \rightarrow get(\rho(r))$ and neither can be $put(r) \rightarrow get(w)$ for any other w in $\mathcal{W}$, since the first write occurring after $put(r)$ initiates a new phase. □

Lemma 4 *For any two reads r^- and r^+, such that $r^- \rightarrow r^+$ and $(\neg\exists$ read r : $r^- \rightarrow r \rightarrow r^+)$, the $(k-1)$-tuples that r^- and r^+ get from the subreads of H differ in at most one bit.*

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

Let r_i denote the ith read and w_i denote the ith write (there are no overlapping writes, as well) in a system execution. We prove the induction basis by showing that for r_1, r_2 it holds that between the first subaction of r_1 and $put(r_2)$ at most one write to the subregisters of H can occur. Suppose, towards a contradiction, that there exist write operations

$w_x, w_{x+1}, \ldots w_{x+q}$ $(q \geq 1)$ whose write suboperations modify subregisters in H and occur between the first subaction of r_1 and $put(r_2)$. Then the following conditions hold:

(i) $put(r_1) \rightarrow get(w_x)$. This is because, due to the initialization, 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.

(ii) $mode(w_x) = 1$ and $mode(w_{x+j}) \neq mode(w_{x+j+1})$ $(0 \leq j \leq q-1)$. This follows from the definition of $mode(w)$, because initially the writer's static variable wm equals $\oplus_{i=1,\ldots,k-1} H_i$ $(= 0)$ and it is modified iff one of the subregisters of H are modified.

But from (i) and from our assumption if follows that $put(r_1) \rightarrow get(w_x) \ldots \rightarrow get(w_{x+q}) \rightarrow put(r_2)$, which implies (from lemma 2) that it should be $mode(w_x) = mode(w_{x+1}) = \ldots = mode(w_{x+q})$. Thus, we have a contradiction to our assumption; therefore, the induction basis is true.

The induction step is proven with similar reasoning and the additional argument (to substitute the initializing conditions) that for each write w it holds that $mode(w) = \oplus_{i=1,\ldots,k-1} H_i$ after the write suboperation of w. □

LEMMA 5 *Let $\mathcal{W}$ be the phase of writes related to a read operation r. Then $mode(r) = mode(\mathcal{W})$ and $view(r) = view(\rho(r))$.*

PROOF. Again, this can be proven by induction on the number of reads, using the following reasoning: From the previous two lemmas it follows that for any read r, while r scans the subregisters of the construction at most two collisions (attempts to concurrently access a subregister) can occur with overlapping writes. These collisions might happen in one of the subreads of H and/or of L^m, where $m = mode(r)$. This implies that r and $\rho(r)$ will (after all the respective updates) have the same values in their local variables h_i, l_i^0 and l_i^1 $\forall i = 1, \ldots, k-1$. □

In order to complete the proof of the atomicity of our construction we will use the following atomicity criterion for single-writer registers (Lamport [14]).

ATOMICITY CRITERION: A register construction is atomic if for any system executions σ the following three conditions are satisfied:

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 $\pi(r_2) \rightarrow \pi(r_1))$.

LEMMA 6 *The protocol satisfies the above atomicity criterion.*

PROOF. Lemma 5 implies that for any read r of a system execution σ of the protocol $\pi(r) = \rho(r)$. Therefore, it suffices to prove that the three conditions of the criterion hold using $\rho(r)$ instead of $\pi(r)$.

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 $\rho(r) \rightarrow w \rightarrow r$. From lemma 5 we have that $mode(\rho(r)) = mode(r) = m$, where $m \in \{0, 1\}$. There are two cases to be considered:

(1)$mode(w) = m$: Then w and $\rho(r)$ are in the same phase of writes, which implies that w will write on a subregister in L^m. This contradicts the definition of $\rho(r)$.

(2)$mode(w) = \overline{m}$: From the protocol we have that $\exists$ a write w' : $\rho(r) \rightarrow w' \rightarrow r$ (w' may equal w) such that w' writes in one of the subregisters in H. This contradicts the definition of $\rho(r)$, since r reads the subregisters in H.

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 where k is not power of 2, then 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.

Theorem 1 *The 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.*

Conclusions

In this work we have shown how a simple "encoding" function can be used in order to simulate a powerful wait-free mechanism. It would be useful to examine whether more sophisticated encoding can be used in order to gain in efficiency in wait-free constructions for various other objects.

Acknowledgment

We are thankful to John Tromp and Paul Vitányi for helpful discussions. We would also like to thank Jaap-Henk Hoepman and the anonymous referees of MFCS 94 for their accurate and useful remarks.

References

1. B. Bloom. "Constructing Two-writer Atomic Registers". *IEEE Transactions on Computers*, **37**:1506–1514, 1988.

2. J.E. Burns and G.L. Peterson. "Constructing Multi-reader Atomic Values From Nonatomic Values". In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, 1987, pp. 222–231.

3. S. Chaudhuri, M.J. Kosa and J.L. Welch. "Upper and Lower Bounds for One-Write Multivalued Regular Registers". In *Proceedings of the 3rd IEEE Symposium on Parallel*

and Distributed Processing, December 1991, pp. 134-141. Also available as TR91-026 from the University of North Carolina at Chapel Hill.

4. S. CHAUDHURI AND J.L. WELCH. "Bounds on the Costs of Register Implementations". In *Proceedings of the 4th International Workshop on Distributed Algorithms*, volume 486 of *Lecture Notes in Computer Science*, Springer-Verlag 1990, pp. 402–421.
5. P.J. COURTOIS, F. HEYMANS AND D.L. PARNAS. "Concurrent Control With Readers and Writers". *Communication of the ACM* **14**(10):667-668, 1971.
6. C. DWORK, M. HERLIHY, S. PLOTKIN AND O. WAARTS. "Time-Lapse Snapshots". In *Proceedings of the First Israel Symposium on the Theory of Computing and Systems*, volume 601 of *Lecture Notes in Computer Science*, Springer-Verlag 1992, pp. 154-170.
7. C. DWORK, O. WAARTS. "Simple and Efficient Bounded Concurrent Timestamping or Bounded Concurrent Timestamp Systems are Comprehensible!" In *Proceedings of the 24th ACM Symposium on Theory of Computing*,1992, pp. 656-666.
8. GOOS KANT AND JAN VAN LEEUWEN. "The File Distribution Problem for Processor Networks". In *Proceedings of the Second Scandinavian Workshop on Algorithm Theory*, volume 447 of *Lecture Notes in Computer Science*, Springer-Verlag 1990, pp. 48-59.
9. M. HERLIHY AND J. WING. "Linearizability: A Correctness Condition for Concurrent Objects". *ACM Transactions on Programming, Languages and Systems* **12**(3):463–492, 1990.
10. A. ISRAELI AND A SHAHAM. "Optimal Multi-Writer Multi-reader Atomic Registers". In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pp. 71-82.
11. P. JAYANTI, A. SETHI AND E.L. LLOYD. "Minimal Shared Information for Concurrent Reading and Writing". In *Proceedings of the 5th International Workshop on Distributed Algorithms*, October 1991, volume 579 of *Lecture Notes in Computer Science*, Springer-Verlag 1992, pp. 212–228.
12. L.M. KIROUSIS, E.KRANAKIS, P.M.B. VITÁNYI. "Atomic Multireader Register" In *Proceedings of the 2nd International Workshop on Distributed Algorithms*, volume 312 of *Lecture Notes in Computer Science*, Springer-Verlag 1987, pp. 278–296.
13. L.M. KIROUSIS, P. SPIRAKIS, PH. TSIGAS. "Reading Many Variables in One Atomic Operation: Solutions With Linear or Sublinear Complexity". In *Proceedings of the 5th International Workshop on Distributed Algorithms*, October 1991, volume 579 of *Lecture Notes in Computer Science*, Springer-Verlag 1992, pp. 229–241. Also to appear in *IEEE Transactions on Parallel and Distributed Systems.*
14. L. LAMPORT. "On Interprocess Communication, Part I: Basic Formalism, Part II: Basic Algorithms". *Distributed Computing*, **1**:77-101, 1986.
15. M. LI, J.TROMP AND P.M.B. VITÁNYI. "How to Construct Concurrent Wait-free Variables". Technical Report CS-8916, CWI, Amsterdam, April 1989. See also: pp. 488–505 in *Proceedings of the International Colloquium on Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, Springer-Verlag, 1989.
16. R. NEWMAN-WOLFE. "A Protocol for Wait-free, Atomic, Multi-reader Shared Vari-

ables". In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, 1987, pp. 232–248.

17. G.L. PETERSON AND J.E. BURNS. "Concurrent Reading While Writing II: The Multi-writer Case". In *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, 1987, pp. 383–392.

18. G.L. PETERSON. "Concurrent Reading While Writing". *ACM Transactions on Programming Languages and Systems* **5**(1):46–55, 1983.

19. R. SCHAFFER. "On the Correctness of Atomic Multi-writer Registers". Technical Report MIT/LCS/TM-364, MIT Lab. for Computer Science, June 1988.

20. A.K. SINGH, J.H. ANDERSON AND M.G. GOUDA. "The Elusive Atomic Register Revisited". In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, 1987, pp. 206–221.

21. J. TROMP. "How to Construct an Atomic Variable" In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, volume 392 of *Lecture Notes in Computer Science*, Springer-Verlag 1989, pp. 492–302.

22. K. VIDYASANKAR. "Converting Lamport's Regular Register to Atomic Register". *Information Processing Letters* **28**:287–290, 1988.

23. K. VIDYASANKAR. "An Elegant 1-Writer Multireader Multivalued Atomic Register". *Information Processing Letters* **30**:221–223, 1989.

24. K. VIDYASANKAR. "Concurrent Reading While Writing Revisited". *Distributed Computing* **4**:81–85, 1990.

25. P. VITÁNYI AND B. AWERBUCH. "Atomic Shared Register Access by Asynchronous Hardware". In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, 1986, pp. 233–243.