Reading Many Variables in One Atomic Operation Solutions With Linear or Sublinear Complexity[1]

Lefteris M. Kirousis[2,3,5], Paul Spirakis[2,3,4,5], Philippas Tsigas[2,3,5]

Abstract

We address the problem of reading more than one variables (components) $X_1, \ldots, X_c$, all in one atomic operation, by *only one* process called the reader, while each of these variables are being written by a set of writers. All operations (i.e. both reads and writes) are assumed to be totally asynchronous and wait-free. For this problem, only algorithms that require at best quadratic time and space complexity can be derived from the existing literature (the time complexity of a construction is the number of sub-operations of a high-level operation and its space complexity is the number of atomic shared variables it needs). In this paper, we provide a deterministic protocol which has linear (in the number of processes) space complexity, linear time complexity for a read operation and constant time complexity for a write. Our solution does not make use of time-stamps. Rather, it is the memory location where a write writes that differentiates it from the other writes. Also, introducing randomness in the location where the reader gets the value it returns, we get a conceptually very simple probabilistic algorithm. This algorithm has an overwhelmingly small, controllable probability of error. Its space complexity as well as the time complexity of a read operation are both sublinear. The time complexity of a write is constant. On the other hand, under the Archimedean time assumption, we get a protocol whose both time and space complexity do not depend on the number of writers but are linear in the number of components *only* (the time complexity of a write operation is still constant).

INDEX TERMS: Atomic Asynchronous Registers, Composite Registers, Distributed Algorithms, Linearazability, Probabilistic Protocols, The Readers-Writers Problem.

[1]This research was partially supported by the ESPRIT Basic Research Program of the EC under contracts no. 7141 (project ALCOM II) and no. 6019 (project Insight II). This work appeared in *IEEE Transactions on Parallel and Distributed Systems*, **5**(7), pp. 688-696, July 1994. A preliminary version of this paper appeared in the *Proceedings of the Fifth International Workshop on Distributed Algorithms (WDAG'91), Lecture Notes in Computer Science Vol.579* (Springer-Verlag), pp. 229-241, 1992.

[2]Department of Computer Engineering and Informatics, University of Patras, Rio, 265 00 Patras, Greece.

[3]Computer Technology Institute, P.O. Box 1122, 261 10 Patras, Greece.

[4]Courant Institute of Mathematical Sciences, NYU, New York, N.Y. 10012, U.S.A.

[5]E-mail addresses: ⟨lastname⟩@grpatvx1.bitnet

1 Introduction

A **shared register** is an abstract data structure shared by a number of asynchronous concurrent processes which perform either read or write operations. We adopt the model where each process is assumed to execute either only read operations—it is then called a **reader**—or only write operations—it is then called a **writer**. Operations by the same process are assumed to be executed sequentially. An implementation (construction) of a register consists of: (i) protocols for the execution of an operation (read or write) by a process (ii) a data structure consisting of memory cells, called subregisters and (iii) a set of initial values of the subregisters. The execution of a protocol by a process involves a number of both read and write operations, called **sub-operations**, on the subregisters. To distinguish operations on the register from sub-operations on the subregisters, we sometimes call the former **high-level** operations. An implementation is **wait-free** if it guarantees that any process will complete an operation in a finite number of steps (i.e., sub-operations) independent of the execution speeds of the other processes. Obviously, the wait-free condition rules out many conventional algorithmic techniques, such as busy-waiting, conditional waiting or critical sections. The basic correctness condition for such an implementation is *linearizability*, i.e. although concurrent operations by processes may overlap in time, each one of them appears to take effect instantaneously, in an order that preserves the operations' semantics (see [7]). Such an implementation is called **atomic**.

In this paper we study a type of register called **composite register**. A composite register is a register partitioned into a number of **components**, $X_1, \ldots, X_c$. A high-level operation on such a register either writes a value to *one* of the components or reads the values of *all* the components (the components should not be confused with the subregisters used in an implementation). A composite register is characterized by the number of readers that may concurrently read the composite register, the number of writers that may concurrently write to the same component, the number of components, and the number of bits a component is allowed to hold. The general case of a composite register is the case of n-reader, m-writer per component, c-component, b-bit composite register. The problem which we study in this paper is the implementation of a *single-reader*, m-writer per component, c-component, b-bit composite register using as subregisters atomic, single-component registers that are allowed to hold a bounded—i.e, independent of the number of operations—number of bits. By our assumption, no concurrent high-level read operations are allowed. There is an extended literature on implementing our building blocks, the single-component, atomic registers, from simpler, i.e., single-bit, single-writer, single-reader, single-component, subregisters satisfying only minimal correctness conditions. See, e.g., [3] for a reference list.

The time complexity of a construction implementing a composite register is the number of sub-operations of an operation, while the space complexity is the number of atomic, single-component subregisters used by the construction. We measure them as a function of the number of the processes which share the composite register.

Afek *et al.* [2] and Anderson [3] have previously given wait-free constructions for the multireader, multiwriter, c-component, b-bit composite register. Their time complexity as well as their space complexity are at best quadratic as a function of the number of processes. The complexities do not improve if we assume the existence of only one reader.

In this paper, we first give a conceptually very simple wait-free construction for the *single-reader*, multiwriter per component case assuming that we have an *unbounded* (but linearly dependent on the total number of operations performed) number of memory locations (sub-

registers). In this first construction, the number of sub-operations of a high-level operation is unbounded and moreover, one subregister is assumed to hold an unbounded (but logarithmically dependent on the total number of operations performed) number of bits.

We then show how to "recycle" the memory locations in order to obtain a deterministic protocol that uses only a bounded number of subregisters. Our bounded construction has linear space complexity, linear time complexity for a read operation and constant time complexity for a write operation (the building blocks, in the bounded case, are assumed to be single-component, atomic registers so that each may hold either a value from the domain of values allowed to appear on the components of the register or, alternatively, an integer not exceeding a constant multiple of the number of writers per component). We believe that the tool of using uboundedly many memory locations is stronger than the method of unbounded time-stamps (for constructions with unbounded time-stamps see [5] and [8]).

Moreover, introducing randomness in the choice of the memory location recycled by a read, we obtain a conceptually very simple probabilistic protocol. If m is the number of writers per component, c is the number of components, and l and q are constants that can be chosen by the algorithm designer, then the space complexity of our probabilistic algorithm is $O(lc)$. The time complexity of a read operation is $O(lc)$, whereas the time complexity of a write operation is $O(q)$. Finally, there is a $O(mc(q/l)^q)$ probability of error. Our randomized protocol works even if the adversary is assumed to observe a random bit the moment it is generated (this is the strong model for an adversary assumed in [1] and [4]).

Randomized algorithms that, as the one in this paper, allow the possibility of error (i.e., Monte Carlo algorithms) may have important drawbacks when applied to shared-memory data structures. However, we believe that they might be interesting not only because the probability of error is overwhelmingly small and controllable—an important factor *per se*—but also because they may pave the way for new Las Vegas randomizations (i.e., no-error randomizations, where, however, complexity bounds are probabilistic). For error-free, randomized protocols with finite expected time complexity see [6].

Finally, under the Archimedean Time assumption, i.e. assuming that there are fixed, known upper and lower bounds for the ratio of the execution rates of the processes (limited asynchrony), we give a protocol with space and time complexities that do not depend on the number of processes but are linear in the number of components *only* (the time complexity of a write is still constant). Notice that this assumption does not imply any restriction on the idle time intervals between operations.

Our notations and definitions closely follow the existing literature (see [3] or [2]). For a formalization we refer to [9] and [7]. Our notation is compatible with these formalisms. We assume that for each operation O there exists a time interval $[s_O, f_O]$ called its **duration**. The points s_O and f_O are the starting and finishing times, respectively of O. We assume that there is a precedence relation on operations which is a strict partial order (denoted by '$\rightarrow$'). For two operations a and b, $a \rightarrow b$ means that operation a ends before operation b starts. If two operations are incomparable under $\rightarrow$, they are said to **overlap**. Since we have assumed that there is only one reader, all read operations are comparable under $\rightarrow$. The protocols, apart from the shared variables, make use of **local** variables as well (these cannot be shared by concurrent processes). The local variables are assumed to retain their values between invocations of the corresponding procedures (in the programming languages literature, the term *static* is sometimes used for such variables). We adopt the convention to denote shared variables with capital letters and local variables with lower case letters.

A **reading function** π_k for a component k is a function that to each high-level read operation r assigns a high-level write operation w on component k such that the value returned by r for the component k is the value written by w. Similarly for a subregister R, a reading function π_R is a function that for each read sub-operation r on R assigns a write sub-operation w on R such that the value returned by r is the value written by w. It is assumed that for each subregister there exists a write sub-operation which initializes the subregister, i.e., precedes all other sub-operations on the subregister.

A **run** (or **history**) is an execution of an arbitrary number of operations according to the respective protocols. Formally, a run is atomic if the partial order $\rightarrow$ on its operations can be extended to a *total* strict order $\Rightarrow$ and if for each component X_k there is a reading function π_k such that for all high-level reads r: (i) $\pi_k(r) \Rightarrow r$ and (ii) there is no write w on X_k such that $\pi_k(r) \Rightarrow w \Rightarrow r$. A construction is atomic if all its runs are atomic. We assume all subregisters to be atomic, therefore we can assume that the precedence relation $\rightarrow$ is total when restricted to sub-operations on a single subregister (alternatively, we assume that all sub-operations are instantaneous—i.e., their duration intervals are singletons).

One obviously necessary condition for a composite register to be atomic is that for any read r and for any component X_k, it is not the case that $r \rightarrow \pi_k(r)$ (indeed, otherwise the extension of $\rightarrow$ to a total order respecting the reads would be impossible). All our constructions will satisfy this condition for trivial to check reasons. For notational convenience, we call registers satisfying this condition **normal**.

2 An Atomicity Criterion

For the case of a single-reader (where we do not have overlapping high-level reads) we have the following criterion for atomicity of a composite register:

Lemma 1 *A construction of a normal composite register is atomic if and only if for each component X_k, the write operations to it can be serialized by a strict total order $\Rightarrow_k$ compatible with the precedence relation $\rightarrow$ and such that the following two conditions hold:*

1. *Each $\Rightarrow_k$ is compatible with the respective reading function π_k, i.e. for each read r, it is not the case that there is a write operation w on X_k so that $\pi_k(r) \Rightarrow_k w \rightarrow r$. Moreover, for any two reads r and s and for any component X_k, it is not the case that: $r \rightarrow s$ and $\pi_k(s) \Rightarrow_k \pi_k(r)$.*

2. *For any two different components X_k and X_l and for any read r, it is not the case that there are write operations v and w on X_k and X_l respectively such that*

$$\pi_k(r) \Rightarrow_k v \rightarrow w \stackrel{=}{\Rightarrow}_l \pi_l(r),$$

where $w \stackrel{=}{\Rightarrow}_l \pi_l(r)$ means that either $w \Rightarrow_l \pi_l(r)$ or $w = \pi_l(r)$.

This lemma is essentially the restriction to the single-reader case of the atomicity criteria mentioned in [3], and so we omit its proof.

Based on the above, we obtain the following basic lemma which gives *sufficient* conditions for atomicity that refer to *each component separately*. All our algorithms satisfy the conditions of this basic lemma. Therefore, our algorithms not only implement an atomic register but

have stronger, in general, properties described by these conditions. Intuitively, the sufficient conditions of the basic lemma require that the write operations of each component can be consistently serialized so that Condition 1 of Lemma 1 is satisfied, and moreover if a write operation w starts after the start of a read operation r, or if w follows (in the serialization of the operations of its component) another write operation that starts after the start of r, then r does *not* read the value written by w. Formally:

Basic Lemma *A construction of a normal composite register is atomic if for each component X_k, the write operations to it can be serialized by a strict total order $\Rightarrow_k$ that is compatible with the precedence relation $\rightarrow$ and such that the following two conditions hold:*

1. *Each $\Rightarrow_k$ is compatible with the respective reading function π_k, i.e. for each read r, it is not the case that there is a write operation w to X_k so that $\pi_k(r) \Rightarrow_k w \rightarrow r$ and moreover, for any two reads r and s and for any component X_k, it is not the case that: $r \rightarrow s$ and $\pi_k(s) \Rightarrow_k \pi_k(r)$.*

2. *For any read r and for any component X_l, if a is either the first sub-operation of $\pi_l(r)$ or the first sub-operation of any write operation w to X_l for which $w \Rightarrow_l \pi_l(r)$, and if b is the first sub-operation of r then a and b take place on the same (atomic) subregister and a precedes b.*

Proof It suffices to prove the second condition of Lemma 1. Indeed, let X_k and X_l be two distinct components. Suppose, towards a contradiction, that there is a v and a w on X_k and X_l, respectively, such that:

$$\pi_k(r) \Rightarrow_k v \rightarrow w \stackrel{=}{\Rightarrow}_l \pi_l(r).$$

Then, since by hypothesis the first sub-operation of w precedes the first sub-operation of r, we get that $\pi_k(r) \Rightarrow_k v \rightarrow r$, a contradiction. □

3 The Deterministic Approach

3.1 Unbounded Memory-Space

In this subsection we are going to describe a single-reader, multiwriter per component construction that uses unbounded memory-space (i.e. the number of subregisters used may be equal to the number of operations to be performed). In the next subsection then, we show how to "recycle" the memory space in order to obtain a construction with bounded space, i.e., independent of the number of operations (actually, the space will be linear in the number of writers). Let us point out that in the unbounded memory-space construction, there is a subregister whose values are addresses of memory locations. Therefore, this subregister must be assumed to hold an unbounded number of bits. This is not the case in the bounded memory-space construction, where there is only a bounded number of addresses. In the unbounded space construction, the number of sub-operations of a high-level read operation is unbounded.

The architecture of our unbounded construction is as follows: For each component $k = 1, \ldots, c$, we introduce an unbounded number of subregisters $ML[k][l]$, $l = 0, \ldots, \infty$ which are written to by the writers of the corresponding component and are read by the reader. We call these subregisters **memory locations**. The second index of each memory location $ML[k][l]$ is its address (the first indicates the corresponding component). A memory location holds a

```
var PTR : integer; ML : array[1..c][0..∞] of valtype; /*Shared variables declaration*/

procedure reader /*returns array[1..c] of valtype*/
var ptr, b : integer; a : array[1..c] of valtype;
begin
    write ptr + 1 to PTR; ptr := ptr + 1;
    for k := 1 to c do
        b := ptr;
        repeat
            b := b − 1; read a[k] from ML[k][b];
        until a[k] ≠nil; od;
    return (a[1], ..., a[c]);
end

procedure writer /*writes u : valtype on component k*/
var w-ptr : integer; u : valtype;
begin
    read w-ptr from PTR; write u to ML[k][w-ptr];
end
```

Figure 1: The unbounded memory-space protocol.

value that belongs either to the set of values of the corresponding component or is a special new value denoted by **nil**. The type of all these values is denoted by *valtype*. We call them **component values**. Initially, the subregisters $ML[k][l]$ for $k = 1 \ldots, c$ and $l = 1, \ldots, \infty$ hold the value **nil**, while the subregisters $ML[k][0]$, $k = 1, \ldots, c$ hold a value from the set of values of the corresponding component. Moreover, we introduce a subregister PTR which holds as value an integer (a pointer to a memory location). This subregister can be written to by the reader and can be read by all writers. It is initialized with the value 0.

In the protocol the reader is the controller: it is the one who determines where the writers must write. All that a writer has to do is to write its value to the memory location forwarded by the reader through a pointer. More specifically, the protocol works as follows: A writer first reads PTR and then writes its value to the memory location of the corresponding component that is pointed to by PTR. The reader, on the other hand, first increments PTR by one; stores its new value into a local variable *ptr* and then for each component $k = 1, \ldots, c$ gets the value to be returned by reading $ML[k][ptr - 1], \ldots, ML[k][0]$ in this order until it gets a value which is not **nil**. The protocol is given formally in Figure 1. The reader, by forwarding to the writer, with its very first sub-operation, a new subregister, which it does not use again during the current read, it succeeds to avoid reading values written by write operations that started after its own starting point. Moreover, the reader, by scanning the subregisters in the reverse order from the one that they were forwarded in previous operations and by returning the first "non-empty" value, it achieves to return nonoverwritten values.

Correctness Proof We will show that the above construction satisfies the two conditions of the Basic Lemma. To show that Condition 1 of the Basic Lemma is satisfied, define the relation $\Rightarrow_k$ between writes on the same component as follows: $w \Rightarrow_k v$ if and only if either w and v write their value to the same memory location and the last sub-operation of w precedes the last sub-operation of v, or w writes its value to a memory location with address less than the address of the corresponding memory location of v. It is clear that thus Condition 1 is satisfied. Also, Condition 2 of the Basic Lemma is satisfied. Indeed, both $\pi_k(r)$ and r have their first sub-operation performed on PTR. If the first sub-operation of $\pi_k(r)$ followed that of r, then $\pi_k(r)$ would write its value to a memory location not visited by r. We get a similar contradiction if there is a w such that $w \Rightarrow_k \pi_k(r)$ and the first sub-operation of r precedes that of w. □

3.2 Bounded Memory-Space

In this subsection, we will show how to transform the unbounded space protocol of the previous subsection into one that uses bounded space only.

First observe that both conditions of the Basic Lemma, through which we prove the correctness of our protocols, refer to each component separately (i.e., no reference to two components is made in any one of the conditions, as is the case, e.g., in Condition 2 of Lemma 1). This property of the Basic Lemma allows us, without loss of generality and for reasons of simplicity, to present our protocols considering only one component. (Notice that the conditions that the protocol must satisfy for each component in order to comply with the requirements of the Basic Lemma are stronger than what is required from a single-component atomic register; therefore, although we assume the existence of only one component, it is not the case that we reduce the problem of multiple components to the single-component case.) So, indices of variables referring to component numbers are not used in this and the next subsection. However for reasons of completeness, in the description of the formal protocol given in Figure 3, we assume that there is an arbitrary number of components.

In the bounded space protocol as well, we are going to keep the role of the reader as the controller of the game. It still is the one who determines the subregister where the writer is going to write. However, because the number of the subregisters must be bounded, instead of forwarding a new subregister each time, the reader has to find an obsolete subregister which will be forwarded to the writer after erasing its contents. We call this procedure of erasing the contents of a subregister and its forwarding to the writer *recycling* of the subregister.

We keep the techniques used in the previous algorithm, that is: (i) The writer writes to the memory location forwarded by the reader. (ii) The reader, by forwarding with its very first sub-operation a recycled subregister, which it is not going to use again during the current read, it succeeds to avoid reading component values written by write operations which start after its own starting point. (iii) The reader in each read operation reads the remaining subregisters—i.e. the entries of the array ML except from the entry corresponding to the subregister $ML[i]$ currently forwarded—in the reverse order from the one that they had been previously forwarded to the writer.

Thus, the problem of designing a correct algorithm that uses a bounded number of subregisters is reduced to the problem of having the reader choose each time a *provably* obsolete subregister for recycling. That means that we have to make sure that the following two conditions are satisfied:

Condition A: A read operation, when recycling, does not erase the component value that it

returns (this is required because this value must be available to the next read as well).

Condition B: A read operation r, when recycling, does not erase a component value written by a subwrite (of a write operation) that follows the first subread of r from an entry of ML of the corresponding component (again, in order to avoid the possibility of erasing a value that must be available to the next read).

The way to guarantee the above conditions is described in the next two subsections. To make the presentation more understandable, we chose to present first the case of a single-writer and in the sequel the case of multiple writers.

3.2.1 The Single-Writer Case

The formal protocol for the single-writer bounded case is given in Figure 2. The initializations of the variables are given at the end of the current Subsection 3.2.1

The reader, in its local memory, maintains an array $ma[1..dim]$ of addresses of memory locations (i.e., the entries of ma are pointers to entries of the array ML). These are the addresses of the memory locations recycled in the last dim read operations, in the order they appear in the array. In other words, by the local array ma, the reader "remembers" the order in which a number of dim memory locations are recycled and forwarded to the writer. For reasons to be explained below, it turns out that the the value of dim must be at least 5. A read operation gets the component value it returns by reading the locations with addresses $ma[dim-1], \ldots, ma[1]$ in this order (i.e. in the reverse order from the one that they were forwarded to the writer, or otherwise in the most-recent-first order) until it finds a value$\neq$**nil**. This different from **nil** value is the component value the reader returns. The procedure recycle that follows these subread operations stores into $ma[dim]$ the address of the *new* location to be recycled (i.e., forwarded to the writer) in the next read operation. This address is chosen from the values of $ma[dim-1], \ldots, ma[1]$ and its contents are erased by the reader (i.e., the value **nil** is written to it). If the address thus chosen is $ma[j]$, the array $ma[j], \ldots, ma[dim]$ is cyclically rotated one position to the left (thus $ma[j]$ becomes the new value of $ma[dim]$ and the previous value of $ma[dim]$ is now stored in $ma[dim-1]$).

To guarantee Condition A, a read operation never recycles the location where it obtained the value it returns. Thus, the value last obtained by the reader remains available for possible future use (otherwise, the next read operation might be left with nothing to read).

To guarantee Condition B a read r must "know" which are the memory locations where a component value by the writer might appear during r and after the first subread of r from ML. These locations should not be recycled. The reader stores the addresses of these not-for-recycling (forbidden) locations into local variables denoted by vb_0 and vb_1. In the next paragraphs we explain how the reader decides which addresses should be stored into vb_0 and vb_1.

First, the shared variable PTR (where, in the unbounded case, the reader writes the memory address it forwards to the writer) now has two fields: one, called $PTR.flagfield$, is a boolean flag; the other, called $PTR.ptrfield$, is a two-entry array storing two memory addresses *both* of which are forwarded to the writer for possible use (however, at each read operation, only one of the two entries of $PTR.ptrfield$ gets a possibly new value). In order to write its component value, the writer chooses one among the two entries in $PTR.ptrfield$ according to the value of $PTR.flagfield$ it reads. Moreover, the writer maintains a shared boolean array $WFLAG$ that is read by the reader.

To be more specific, the writer first reads PTR and copies the value of $PTR.flagfield$ to $WFLAG$. Then it re-reads PTR and moves on to the memory address $PTR.ptrfield[WFLAG]$, where it writes the component value.

On the other hand, the reader, during an operation r, first updates the variable PTR. In $PTR.flagfield$, it writes the complement of the value obtained from $WFLAG$ *in its previus read operation* (at the starting point of r, this complement is available through the reader's local variable $flag$). In $PTR.ptrfield[flag]$, it writes the address that in its previous operation decided to recycle (i.e., the address in $ma[dim]$). The other entry of $PTR.ptrfield$ gets the same value it had before. Then the reader stores the value of $WFLAG$ in its local variable $flag$ and moves on to scan the addresses stored in ma in order to decide, as explained above, which value to return. Finally, it executes the 'recycle' procedure and returns. During the procedure 'recycle', the reader, as explained above, chooses the address $ma[j]$ to be recycled, erases the value of the corresponding memory location and cyclically rotates the array $ma[j], \ldots, ma[dim]$. Then, it complements the value of its local variable $flag$ and updates the values of $vb_{flag}(:= \{ptr[flag], ma[5]\})$ and $ptr[flag](:= ma[5])$.

The alternation of the values of the boolean variables, and the consequent alternation between the two entries of $PTR.ptrfield$, where the writer gets the address it uses in order to write its component value, guarantees that the reader has the correct knowledge about the forbidden addresses which must be stored in vb_0 and vb_1. Indeed, suppose, w.l.o.g., that a read operation r reads at its 'read from $WFLAG$' sub-operation the boolean value 0 and suppose that this value was written to $WFLAG$ by a write operation w. Let w^+ be the write operation immediately following w and let r' be the *last* read operation preceding r that reads at its 'read from $WFLAG$' sub-operation the boolean value 1. Notice that according to the protocol, r chooses for recycling a memory address not in $vb_0 \cup \{ma[i], ma[dim]\}$. Also, vb_0 is last updated during r'. Since the first subread of r from ML (i.e., the subread from $ML[ma[dim-1]]$) follows its subread from $WFLAG$, it can be easily seen that a write operation that writes a component value during r and after the first subread of r from ML must be either (i) w or (ii) w^+ or (iii) a write operation that started after the starting point of r. Also, the write operations (i)–(iii) (given that they finish before the end of r) choose to write their respective component values in addresses obtained from the variable $PTR.ptrfield$ at an instant when this variable carries values written to it by a read operation between r' and r (r' and r included). This is so because, since r' reads the value 1 from $WFLAG$, the 'read from $WFLAG$' sub-operation of r' must precede the 'write to $WFLAG$' sub-operation of w and hence it must precede the w's second reading of PTR as well. Using this last fact and by an easy case analysis, it follows that the write operations (i)–(iii) (given that they they finish before the end of r) choose addresses that are either in vb_0 or are equal to the value that $ma[dim]$ has at the start of r. Therefore the component values of the write operations (i)–(iii) are not erased by r, and so Condition B is satisfied.

Notice that according to the protocol, both vb_0 and vb_1 have at most two elements. Since the address to be recycled must be chosen not to be in the set $vb_{flag} \cup \{ma[i], ma[dim]\}$ and since this set has at most four elements, the value of dim should be at least five.

We have proved that both Conditions A and B are satisfied. Since we have assumed that there is only one writer per component, the write operations on each component are linearly ordered by $\rightarrow$, therefore, it is now easy to show that both conditions of the Basic Lemma are satisfied.

The initialization of the shared variables is the following: $ML[4]$:= a value ($\neq$**nil**) from the set of possible values of the component, and $ML[i]$:=**nil**, for $i = 1, 2, 3, 5$. $PTR.flagfield := 0$

and $PTR.ptrfield := (4,4)$. $WFLAG := 0$. Reader's local variables are initialized as follows: $vb_0, vb_1 := \{4,5\}$, $ma[1], \ldots, ma[5] := 1, \ldots, 5$, $flag := 0$ $ptr := (5,5)$. All other variables are arbitrarily initialized.

3.2.2 The Multiwriter Case

In this case it is assumed that there are m writers that may concurrently attempt modifications to a component. For reasons explained in the introductory paragraphs of Subsection 3.2, we still assume without loss of generality that there is only one component. The idea is to have the reader interact with each writer (of each component) separately, while the order of its actions remain the same as in the single-writer case. This necessitates the extension of the fields $flagfield$ and $ptrfield$ of the shared variable PTR to m-dimensional arrays, each entry of which must contain, exactly as in the single-writer case, guidance information for one of the m writers. However (for each component separately) and independently of the number of writers, the shared array ML and the the local array ma (that correspond to each component) remain of dimension 1. For reasons to be shortly explained though, their length changes from 5 to $2m+3$. The subregister PTR is again one atomic variable and is updated during the first sub-operation of each read. It must be pointed out that during an invocation of the subroutine 'recycle', a unique memory location is recycled and forwarded to all writers. The address of this unique location is written to all entries $PTR.ptrfield[[flagfield][l]][l], l = 1 \ldots m$ ($flag[l]$ is determined by reading $WFLAG[l]$; index l refers to the lth writer of the component). Furthermore, for each writer separately, the reader must know which are the memory locations where this writer might write to, so that it does not recycle them. This is implemented by having the reader keep, for each writer separately, two sets, each having as elements at most two possibly forbidden-to-be-recycled memory addresses—exactly as in the single-writer case. During an invocation of the procedure 'recycle', for each writer, again only one of the two forbidden sets is considered active, according to the value of the corresponding $flag$. It follows that in order to always have a spare memory location to recycle, at least $2m+3$ memory locations should be kept in ma. We give the formal protocol for this case in Figure 3 (in the formal protocol, we assume that the number of components is arbitrary). The initialization of the variables is, for each component and for each writer, analogous to the single-component, single-writer case (with $2m+3$ in place of 5).

Proof of Correctness

Conditions A and B mentioned in the introductory paragraphs of Subsection 3.2 remain true, because the communication (other than reading and writing component values) of the reader with each writer is through separate variables. However, in the multiwriter case, in order to show that the conditions of the Basic Lemma are satisfied, we also need to define a total order $\Rightarrow$ among the write operations of each component. Towards this, we first define the *tag* of a read r to be an integer whose value is equal to the number of read operations that precede r. Whenever a memory location is recycled by a read r whose tag is t, we say that this memory location gets associated with the tag t. This association is kept active until the location is recycled anew; the association is then updated to hold with the tag of the new read invoking the 'recycle' subroutine. Also, we say that a write w is associated with a tag t, if the subregister where w is going to write its component value is associated with the tag t at the moment when the write of this value takes place.

Now, for each pair of write operations w and w' define $w \Rightarrow w'$ if w and w' are associated to tags t and t', respectively, and either (a) $t < t'$ or (b) $t = t'$ and (consequently) w and w' write

```
type Rtype = record flagfield : 0..1; ptrfield :array[0..1] of 1..5; end;
var  PTR : Rtype; WFLAG : 0..1; ML : array[1..5] of valtype;
     /*Shared variables declaration*/

procedure writer /*writes u : valtype*/
var  d, e : Rtype; m : 1..5; u : valtype;
begin
     read d from PTR; write d.flagfield to WFLAG;
     read e from PTR; m := e.ptrfield[d.flagfield]; write u to ML[m];
end

procedure reader /*returns a : valtype*/
var  ptr : array[0..1] of 1..5; flag : 0..1;
     ma : array[1..5] of 1..5; i : 1..5;
     vb_0, vb_1 : set of 1..5;    a : valtype;

     procedure recycle
     var j : 1..5;
     begin
          choose j such that ma[j] ∉ vb_flag ∪ {ma[i], ma[5]};
          rotate one position to the left ma[j], ..., ma[5];
          ML[ma[5]] :=nil; flag := not flag;
          vb_flag := {ptr[flag], ma[5]}; ptr[flag] := ma[5];
     end;

begin
     write (flag, ptr[0..1]) to PTR; i := 5;
     read flag from WFLAG;
     repeat
          i := i − 1; read a from ML[ma[i]];
     until a ≠nil;
     recycle; return a;
end
```

Figure 2: The protocol for the single-writer, single-component, bounded space case.

```
type Rtype = record flagfield : array[1..c][1..m] of 0..1;
                    ptrfield : array[0..1][1..c][1..m] of 1..2m + 3; end;
var  PTR : Rtype; WFLAG : array[1..c][1..m] of 0...1; ML : array[1..c][1..2m + 3] of valtype;
     /*Shared variables declaration*/

procedure reader /*returns array[1..c] of valtype*/
var  vb_0, vb_1 : array[1..c][1..m] of sets of 1..2m+3; i : 1..2m + 3; k : 1..c;
     ptr : array[0..1][1..c][1..m] of 1..2m+3;          flag : array[1..c][1..m] of 0..1;
     ma : array[1..c][1..2m + 3] of 1..2m+3;            a : array[1..c] of valtype;

     procedure recycle(k : 1..c)
     var j : 1..2m + 3;
     begin
          choose j such that ma[k][j] ∉ ⋃_{l=1}^{m} vb_{flag[k][l]}[k][l] ⋃ {ma[k][i], ma[k][2m + 3]};
          rotate one position to the left ma[k][j], ..., ma[k][2m + 3];
          ML[k][ma[k][2m + 3]] :=nil;
          for l := 1 to m do
               flag[k][l] := notflag[k][l];
               vb_{flag[k][l]}[k][l] := {ptr[flag[k][l]][k][l], ma[k][2m + 3]};
               ptr[flag[k][l]][k][l] := ma[k][2m + 3]; od;
     end

begin
     write (flag[1..c][1..m], ptr[0..1][1..c][1..m]) to PTR;
     for k = 1 to c do
          for l := 1 to m do read flag[k][l] from WFLAG[k][l] od;
          i := 2m + 3;
          repeat
               i := i − 1; read a[k] from ML[k][ma[k][i]];
          until a[k] ≠nil;
          recycle(k); od;
     return (a[1], ..., a[c]);
end

procedure writer i /*writes u : valtype on component k*/
var d, e : Rtype; m : 1..2m + 3; k : 1..c; u : valtype
begin
     read d from PTR; write d.flagfield[k][i] to WFLAG[k][i];
     read e from PTR; m := e.ptrfield[d.flagfield[k][i]][k][i]; write u to ML[k][m];
end
```

Figure 3: The bounded memory-space protocol.

their component values to the same subregister and the corresponding subwrite of w precedes the respective subwrite of w' (according to the total order implied by the subregister atomicity).

It is obvious that $\Rightarrow$ is compatible with $\rightarrow$. It is also easy to show that Conditions 1 and 2 of the Basic Lemma are satisfied. So we have proved:

Theorem 1 *A single-reader, c-component, m-writer per component composite register can be constructed using $O(m \cdot c)$ (i.e., linear in the number of processes) atomic, single-reader, multi-writer subregisters and one atomic, multireader, single-writer subregister. The number of steps for a read operation is linear in the number of processes, while a write operation has only four sub-operations.*

4 Sublinear Complexity Solutions under Assumptions

4.1 A Probabilistic Approach

In this subsection, we describe a randomized protocol which will satisfy the atomicity requirements, except that for each high-level read r there is an overwhelmingly small and controllable probability that r will erase a value of a write that, otherwise, might have been read by a later read (a run is atomic if we ignore such erased writes).

The idea is (again) to recycle the memory space (which is assumed bounded). The protocol works essentially as in the deterministic case, except that the value to be written to PTR is chosen randomly rather than through the subroutine 'recycle'. To avoid, with high probability, to recycle a memory location where a pending write operation may write, we assume that there are sufficiently many locations that are candidates for recycling. The number of these locations, l, is determined by the algorithm designer. The protocol is formally given in Figure 4.

Analysis of the protocol's behaviour

The protocol, in order to be correct, must guarantee that Conditions A and B of Subsection 3.2 are satisfied.

Indeed, for the protocol under examination, note that a read operation never recycles the location where it got the value it returns. Thus, Condition A is satisfied. However, Condition B is not deterministically satisfied because it is possible for a read operation r to erase a component value written after the first subread of r from ML by a write operation that started before the start of r (we call such write operations **overlapping erased** writes). A run following our randomized protocol is atomic if we ignore the overlapping erased writes. Notice, however, that in the case of a single component and a single writer on it, for each read there is at most one overlapping write that starts before the start of the read. The reader scans $l-1$ memory locations among which it chooses the one it recycles. It does not choose for recycling the location where it obtains the value that it returns. Therefore the probability of erasing the value of an overlapping write is at most $\frac{1}{l-2}$, where l is the number of memory locations (the value of l is decided by the algorithm designer).

For the c-component, m-writer per component case, the reader chooses independent random numbers in each component's window of memory. Taking into account that for each component there are m locations where an overlapping erased write may appear, we get that the probability for a read r not to erase an overlapping write on any one of the c components is at least $(1-\frac{m}{l-2})^c$. Choosing $l = m \cdot c \cdot \omega + 2$, we get (by the Bernoulli Inequality) that:

```
var PTR : array[1..c] of 1..l; ML : array[1..c][1..l] of valtype; /*Shared variables declaration*/

procedure reader /*returns array[1..c] of valtype*/
var ptr : array[1..c] of 1..l; i : 1..l; ma : array[1..c][1..l] of 1..l; a : array[1..c] of valtype;
begin
    write ptr[1..c] to PTR;
    for k := 1 to c do
        i := l;
        repeat
            i := i − 1; read a[k] from ML[k][ma[k][i]];
        until a[k] ≠nil;
        randomly choose j such that ma[k][j] ∉ {ma[k][i], ma[k][l]};
        rotate one position to the left ma[k][j], ..., ma[k][l];
        ML[k][ma[k][l]] :=nil; ptr[k] := ma[k][l]; od;
    return (a[1], ..., a[c]);
end

procedure writer /*writes u : valtype on component k*/
var w-ptr: array[1..c] of 1..l; u : valtype;
begin
    read w-ptr from PTR; write u to ML[k][w-ptr[k]];
end
```

Figure 4: The probabilistic protocol.

Theorem 2 *Our randomized protocol (for the case of c-components, m-writers per component) has the property that for each read r the probability that r does not erase an overlapping write is at least* $1 - \frac{1}{\omega}$*, where* $\omega = \frac{l-2}{m \cdot c}$ *and l is chosen by the algorithm designer. Moreover, a run is atomic if we ignore the overlapping erased writes.*

Moreover, observe that our randomized protocol works even if the adversary can observe a random bit the moment it is generated. This is so because if the choice of the memory location where a write operation w will write its component value is made after the starting point of a read operation r, then the memory location that w will choose does not depend on the random number generated by the read operation r. If on the other hand, the choice of w is made before the starting point of r, then, obviously, the random number to be generated by r is not known to the operation w at the moment the choice is made.

We can further improve our probabilistic algorithm as follows: the reader instead of keeping one memory address ($ptr[k]$) for each component, keeps a sequence $ptr_1[k], \ldots, ptr_q[k]$ of them, where q is a constant to be chosen by the algorithm designer. These addresses are chosen randomly and uniformly so that they are all different from the location where the read gets the value it returns and from the previous values of the $ptr[k]$'s. On the other hand, a writer of the component k writes its value on the q memory locations it reads from PTR. Now observe that a write can be erased by the one or more reads that overlap its subwrites. By an easy counting argument, it can be proved that for any particular component, the probability for such an error to take place is $O(m(q/l)^q)$ (l is the number of memory locations). Therefore by the Bernoulli Inequality, the probability not to erase an overlapping write on any component is $\Omega((1 - mc(q/l)^q))$. From that we get that:

Theorem 3 *The space complexity of our improved probabilistic protocol as well as the time complexity for a read operation are* $O(lc)$*. The time complexity for a write is* $O(q)$ *and the probability for a read to erase an overlapping write is* $O(mc(q/l)^q)$ *(q and l are chosen by the algorithm designer). A run is atomic if we ignore the overlapping erased writes.*

As a remark, notice that if we choose $q = l/m$ (that makes a read operation mc times slower than a write operation, something to be expected since there is only one reader *vs* mc writers) and if moreover we want a total error expectation of at most ϵ in a number N of read operations (ϵ and N are given), then it is enough to choose $l \geq m + m(\log_m(cN/\epsilon))$. For example, if we have 100 components with 10 writers each, and if we want a total error expectation of .5% in, say 5×10^{14} read operations (if a read operation needs a nanosecond to be executed, then 5×10^{14} of them, by the same reader, need at least fifteen years), then l must be at least 200, so the space complexity of our construction is of the order of 20,000 registers (as it can be easily verified, the constants in the O complexity computations are very small).

4.2 An Approach under the Archimedean Assumption

It has been pointed out (see, e.g., [10] or [11]) that in real distributed systems, it is reasonable to assume that the ratio of the rates of execution of elementary instructions for arbitrary pairs of processes is bounded by a fixed constant. In other words, it is assumed that the clocks of any two processes have a bound on their running rates. Systems complying with such a restriction are called **Archimedean** (this assumption does not imply any restriction on the idle time intervals between two high-level operations by the same process). In this section, we give a protocol for

```
var PTR : array[1..c] of 1..3; ML : array[1..c][1..3] of valtype; /*Shared variables declaration*/

procedure reader /*returns array[1..c] of valtype*/
var ptr : array[1..c] of 1..3; i : 1..3; ma : array[1..c][1..3] of 1..3; a : array[1..c] of valtype;
begin
    write ptr[1..c] to PTR;
    busy wait for A steps;
    for k := 1 to c do
        i := 3;
        repeat
            i := i − 1; read a[k] from ML[k][ma[k][i]];
        until a[k] ≠nil;
        choose j such that ma[k][j] ∉ {ma[k][i], ma[k][3]};
        rotate one position to the left ma[k][j], ..., ma[k][3];
        ML[k][ma[k][3]] :=nil; ptr[k] := ma[k][3]; od;
    return (a[1], ..., a[c]);
end
```

Figure 5: The protocol for the Archimedean case.

a composite register under the Archimedean assumption. Our construction has the interesting property that both its space and time complexity are independent of the number of processes and are both linear only on the number of components. Moreover, the time complexity of a write operation is an absolute constant.

To formalize the above notions, we assume that there is a global time-reference system, which however is not known to the processes (this is not an essential restriction; it is proved in [9] that under some quite general assumptions any system has such a global-time model). Therefore, with every operation (low- or high-level) there is associated a finite time interval, its **duration**. Now, our assumption of Archimedean time states that there is a fixed integer A_0 such that for any two high-level operations a and b and for any time interval I within which a completes the execution of A_0 elementary instructions, if b starts before I does, then b completes the execution of at least one elementary instruction before the end of I. It must be pointed out that by elementary instructions we mean instructions at the lowest level (e.g., assignments of variables, tests, calculations of logical or arithmetical expressions, etc.) Observe, however, that for any particular implementation of subregisters, a constant A can be found, (depending on A_0 and this implementation) such that if within I, a executes A elementary instructions, then b will complete at least one sub-operation (subread or subwrite) before the end of I.

The idea of our construction is the following: As explained in the previous subsection, a basic difficulty for a read r in selecting a memory location to be recycled is to avoid an $ML[x]$ where x is an old value of PTR that a "slow" write w (i.e. one that overlaps r and which started before the start of r) read in the past. If such an x is chosen, then the value of $ML[x]$ can be erased *after* w writes on it, thus the next high-level read may miss values. Notice, however that such a w

must have started before the start of r. So, by the Archimedean-time assumption, if we require from r to do busy-waiting for a sufficient number of its clock ticks, *before* it starts reading the $ML[x]$'s and *after* it has written on PTR, we can guarantee that r will see all writes that are to write on an $ML[x]$ with $x \neq ptr$. We give the formal protocol for the reader in Figure 5. Notice that although there is a busy-wait instruction, the length of this wait is constant (independent of the length of any other operation), therefore the protocol is "wait-free". The protocol for the writer is exactly the same as in the probabilistic case. So, we have:

Theorem 4 *Under the Archimedean assumption, a single-reader, c-component, m-writers per component composite register can be constructed with time and space complexities independent of the number of processes. Specifically, the number of subregisters is* $3c + 1$*, the number of sub-operations of a read operation is at worst* $2c + 1$*, while a write has only two sub-operations.*

5 Conclusion

We have dealt with the problem of designing objects (data structures) shared by asynchronous, wait-free readers and writers. We examined the case of a shared array that must be atomically read by a *single* reader while each entry of the array is written by a set of writers. Constructions for the more general problem of multiple readers were known. However, the complexity of the extant solutions, even for the case of a single-reader, is at best quadratic. In this paper, we gave a solution that for the *single*-reader case has linear space complexity, linear time complexity for a read, and constant time complexity for a write. Moreover, again for the single-reader, multiwriter per component case, we gave probabilistic algorithms with very small, controllable probability of error. These algorithms have sublinear space complexity and also sublinear time complexity for a read. The time complexity for a write is still constant. Finally, we examined a model of limited asynchrony known as Archimedean. For this model, we gave a protocol whose both time and space complexity do not depend on the number of processes. They are linear in the number of entries of the shared array (the time complexity of a write is still constant).

Acknowledgments

We thank Marina Papatriantafillou for her help in improving the presentation of this paper. We also thank the referees for their helpful comments.

References

[1] K. Abrahamson, "On achieving consensus using a shared memory," *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing* (ACM Press, New York), pp. 291–302, 1988.

[2] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt and N. Shavit, "Atomic snapshots of shared memory," *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing* (ACM Press, New York), pp. 1–14, 1990.

[3] J.H. Anderson, "Composite registers," *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing* (ACM Press, New York), pp. 15–30, 1990.

[4] J. Aspnes and M. Herlihy, "Fast randomized consensus using shared memory," *Journal of Algorithms* **11**, pp. 441–461, 1990.

[5] J. Aspnes and M. Herlihy, "Wait-free data structures in the asynchronous PRAM model," *Proceedings of the 2nd Annual ACM Symposium on Parallel Architectures and Algorithms* (ACM Press, New York), pp. 340–349, 1990.

[6] M. Herlihy, "Randomized wait-free concurrent objects," *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing* (ACM Press, New York), pp. 11–21, 1991.

[7] M.P. Herlihy and M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems* **12**, pp. 463–492, 1990.

[8] L.M. Kirousis, P. Spirakis, and Ph. Tsigas, "Simple atomic snapshots: a solution with unbounded time stamps", *Advances in Computing and Information — ICCI '91, Proceedings of International Conference on Computing and Information* (Lecture Notes in Computer Science, Springer-Verlag, Berlin), pp. 582–587, 1991.

[9] L. Lamport, "On interprocess communication, part i: basic formalism, part ii: basic algorithms," *Distributed Computing* **1**, pp. 77–101, 1986.

[10] J.H. Reif and P. Spirakis, "Real-time synchronization of interprocess communication," *ACM Transactions on Programming Languages and Systems* **6**, pp. 215–238, 1984.

[11] P. Vitányi, "Distributed elections in an Archimedean ring of processors," *Proceedings of the 16th Annual ACM Symposium on Theory of Computing* (ACM Press, New York), pp. 542–547, 1984.