Step Optimal Implementations of Large Single-Writer Registers*

Tian Ze Chen[1] **and Yuanhao Wei**[2]

1 **Department of Computer Science, University of Toronto, Toronto, Canada**
`tianze.chen@mail.utoronto.ca`
2 **Department of Computer Science, University of Toronto, Toronto, Canada**
`yuanhao.wei@mail.utoronto.ca`

Abstract

We present two wait-free algorithms for simulating an ℓ-bit single-writer register from k-bit single-writer registers, for any $k \geq 1$. Our first algorithm has $\Theta(\ell/k)$ step complexity for both READ and WRITE and uses $\Theta(4^{\ell-k})$ registers. An interesting feature of the algorithm is that READ operations do not write to shared variables. Our second algorithm has $\Theta(\ell/k + (\log n)/k)$ step complexity for both READ and WRITE, where n is the number of readers, but uses only $\Theta(n\ell/k + n(\log n)/k)$ registers. Combining both algorithms gives an implementation with $\Theta(\ell/k)$ step complexity using $\Theta(n\ell/k)$ space for any $1 \leq k < \ell$.

We also prove that any implementation with $O(\ell/k)$ step complexity for READ requires $\Omega(\ell/k)$ step complexity for WRITE. Since reading ℓ-bits requires at least $\lceil \ell/k \rceil$ reads of k-bit registers, our lower bound shows that our implementation is step optimal.

1998 ACM Subject Classification E.1 Distributed Data Structures, F.1.2 Parallelism and Concurrency

Keywords and phrases atomic register, regular register, wait-free implementation, single writer, optimal

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.32

1 Introduction

A register is a fundamental object that supports READ and WRITE operations. Implementing large ℓ-bit registers from small k-bit registers in a wait-free manner is a classic problem in distributed computing. We consider this problem for atomic single-writer registers shared by n readers. This problem arises naturally in practice when ℓ-bits need to be written atomically on a system that provides only k-bit single-writer registers.

We define the *space complexity* of an implementation of ℓ-bit registers from shared k-bit registers to be the number of k-bit registers that it uses, and we define the *step complexity* to be the number of **read** and **write** operations on these k-bit registers. Note that $\lceil \ell/k \rceil$ **read** steps are required for an ℓ-bit READ operation. Also, any implementation requires $\lceil \ell/k \rceil$ space, since 2^ℓ different values need to be represented.

In 1983, Peterson [9] presented an implementation with $\Theta(\ell/k)$ step complexity for both READ and WRITE. Peterson's implementation has space complexity $\Theta(n\ell/k)$ and works for all $k \geq 1$.

Later, Larsson et al. [8] improved the step complexity of WRITE to $\Theta(n + \ell/k)$, but they used SWAP and FETCHANDOR as primitives.

* This work was supported by the Natural Science and Engineering Research Council of Canada.

Algorithm	Read	Write	Space	Restriction
Peterson (1983)	$\Theta(\ell/k)$	$\Theta(n\ell/k)$	$\Theta(n\ell/k)$	None
Chaudhuri, Kosa, Welch (2000)	$\Theta(4^\ell)$	1	$\Theta(4^\ell)$	None
Aghazadeh, Golab, Woelfel (2014)	$\Theta(\ell/k)$	$\Theta(\ell/k)$	$\Theta(n\ell/k)$	$k \in \Omega(\log n)$
This Paper	$\Theta(\ell/k)$	$\Theta(\ell/k)$	$O(n\ell/k)$	None

Figure 1 Step complexities for implementations of an atomic ℓ-bit single-writer register from atomic k-bit single writer registers.

In 1991, Vidyasankar [10] showed that an atomic ℓ-bit register can be implemented from two regular ℓ-bit registers and one atomic binary register. His WRITE algorithm performs 2 regular writes and 2 atomic writes, and his READ operation performs 2 regular reads and 1 atomic read in the worst case.

Later, Chaudhuri and Welch [4] presented an implementation of regular ℓ-bit registers from regular binary registers with step complexity $\Theta(\ell)$ for both READ and WRITE, using $\Theta(2^\ell)$ space.

In 2000, Chaudhuri, Kosa and Welch [3] presented an atomic ℓ-bit register implementation from atomic binary registers in which each WRITE operation performs a single step. However, the step complexity of their READ operation and their space complexity are both $\Theta(4^\ell)$.

Recently, Aghazadeh, Golab and Woelfel [1] implemented an ℓ-bit *multi-writer* register from k-bit *multi-writer* registers with step complexity $\Theta(\ell/k)$ for both READ and WRITE. Their implementation uses $\Theta(n^2\ell/k)$ registers and requires that $k \in \Omega(\log n)$.

The table in Figure 1 summarises the existing implementations. Peterson's, Chaudhuri and Welch's, and Aghazadeh, Golab and Woelfel's implementations are described in more detail in Section 3. Also in Section 3, we show how to modify Aghazadeh, Golab and Woelfel's implementations to obtain an implementation of an ℓ-bit *single-writer* register from k-bit *single-writer* registers with $\Theta(\ell/k)$ step complexity and $\Theta(n\ell/k)$ space complexity, provided that $k \in \Omega(\log n)$.

In this paper, we present an implementation of an atomic ℓ-bit single-writer register from atomic k-bit single-writer registers with $\Theta(\ell/k)$ step complexity that works for all $k \geq 1$. Our implementation uses $O(n\ell/k)$ registers, which is the same as Peterson's implementation and the single-writer variant of Aghazadeh et al.'s implementation.

We show that our implementation is optimal by proving that any implementation with $O(\ell/k)$ step complexity for READ requires $\Omega(\ell/k)$ step complexity for WRITE.

Our register implementation is the composition of a *tree based* implementation and a *buffer based* implementation. Our tree based implementation has $\Theta(\ell/k)$ step complexity and uses $\Theta(4^{\ell-k})$ registers. An interesting feature is that readers never write to shared registers. This means that helping techniques, such as announcing operations and handshaking, are not used. This implementation can be modified to implement a modulo m counter from k-bit *single-writer* registers that supports a single incrementer and any number of readers. Our counter uses $\Theta(m/2^k)$ registers and has $\Theta((\log m)/k)$ step complexity for READCOUNTER and INCREMENT.

Our buffer based implementation uses this counter as well as known techniques, such as announcement arrays, round-robin helping, and handshake objects, to obtain an implementation with step complexities $\Theta(\ell/k + (\log n)/k)$, while using only $\Theta(n\ell/k + n(\log n)/k)$ registers.

When $\ell \leq \lceil(\log_2 n)/2\rceil$, our tree based implementation has optimal step complexity and uses $O(n/4^k)$ registers. When $\ell > \lceil(\log_2 n)/2\rceil$, our buffer based register implementation has

optimal step complexity and uses $\Theta(n\ell/k)$ registers. Combining these two algorithms gives a step optimal implementation using $O(n\ell/k)$ registers for any $1 \leq k < \ell$.

2 Preliminaries

A *single-writer register* R is a shared register where only one process can perform WRITE operations and any number of processes can perform READ operations. We say that a process owns R if it can write to R.

A *single-incrementer modulo m counter* is a shared modulo m counter where only one process can perform INCREMENT operations and any number of processes can perform READCOUNTER operations. We say that a process owns the counter if it can increment the counter. The counter can take on values from 0 to $m-1$.

We will work in the standard asynchronous shared memory model with n readers p_0, p_1, $\ldots$, p_{n-1} and one writer, which communicate through k-bit registers. Processes may fail by crashing under our model.

In our model, an *execution* is an alternating sequence of *configurations* and *steps* C_0, e_1, C_1, e_2, C_2, $\ldots$, where C_0 is an *initial configuration*. Each step is either a **read** or **write** of a k-bit register. Configuration C_i consists of the state of every register and every process after the step e_i is applied to configuration C_{i-1}. For any two configurations C and C', we use $C \to C'$ to denote that C precedes C' in the execution.

If $C \to C'$, the *execution interval* $[C, C']$ is the set of all configurations and steps between C and C', inclusive. Similarly, the *execution interval* of an operation is the set of all configurations and steps from the first step of that operation to the configuration immediately after the last step of that operation. The execution interval for an *incomplete operation* is the set of all configurations and steps starting from the first step of that operation. Two execution intervals *intersect* if they have a common configuration or step.

We say an object is *atomic* (or equivalently, its implementation is *linearizable* [5]) if, for every possible execution and for each operation on that object in the execution, we can pick a configuration in its execution interval to be its linearization point, such that the operation appears to occur instantaneously at this point. In other words, all operations on the object must behave as if they were performed sequentially, ordered by their linearization points.

We say a register is *regular* if the value returned by each READ is either the value written by the last WRITE operation completed before the first step of the READ or the value written by a WRITE operation concurrent with the READ operation. Note that every atomic register is also regular.

To emphasize the important distinction between atomic and regular registers, consider the scenario where a WRITE operation is concurrent with two READ operations by the same process. Suppose the WRITE operation changed the register from value a to value b. If the register is regular, then each read operation is allowed to return either a or b. However, if the register is atomic, then the second read operation must return b if the first READ operation returns b.

All implementations that we discuss will be *wait-free*. This means that each operation by any process p_i is guaranteed to complete within a finite number of steps by p_i. The *step complexity* of an operation O is the maximum number of steps, over all possible executions, that the process which invoked O performs in the execution interval of O. The *space complexity* of an implementation is defined to be the number of shared registers that it uses.

3 Related Work

In this section, we will briefly review a few existing implementations. Our implementation will build upon these implementations.

Both Peterson's and Aghazadeh et al.'s implementations represent an ℓ-bit value using an array of $\lceil \ell/k \rceil$ registers, each containing k-bits. This construction is called a *buffer*.

In Peterson's implementation, there are two global buffers, $G[0]$ and $G[1]$. Each reader also has its own message buffer. WRITE operations alternate between writing their values to $G[0]$ and $G[1]$. The writer flips a binary register V to indicate the currently active buffer. After flipping V, the writer then checks for help requests from each reader and writes the current value into the message buffers of the readers that requested help. Finally, the writer acknowledges those help requests. At a high-level, a READ operation performs the following steps: request help from the writer, read $G[V]$, and check for an acknowledgement. If no acknowledgement was received, then the value that it read from $G[V]$ has not been overwritten, so it returns what it read. Otherwise it uses the value that the writer put in its message buffer. Essentially, the idea is that if the reader is fast, it can return what it read from G, and if the reader is slow, it will be helped by the writer. Peterson's implementation has step complexities $\Theta(\ell/k)$ and $\Theta(n\ell/k)$ for READ and WRITE respectively, and space complexity $O(n\ell/k)$.

Aghazadeh et al.'s implementation is more complex and requires a novel garbage collection scheme, which they introduced. They implemented a large *multi-writer* register from small *multi-writer* registers. Their implementation achieves $\Theta(\ell/k)$ step complexity for WRITE by having each writer help readers in a round-robin fashion. Thus, only one reader is helped during each WRITE. Since helping is less frequent in their algorithm, they use an array G of $n(8n+1)$ buffers, instead of 2 buffers, to ensure that a reader is helped before the value it wants to read is overwritten. This means that they require $\Theta(\log n)$ size registers to store pointers to elements in G, which leads to the requirement that $k \in \Omega(\log n)$.

Their implementation can be converted into an implementation of an ℓ-bit single-writer register from k-bit single-writer registers. When there is only one writer, each multi-writer register in their implementation is shared by only two processes. Israeli and Shaham [6] presented an implementation of a multi-writer register shared by p processes from single-writer registers of the same size with $\Theta(p)$ step complexity for READ and WRITE. We can use this implementation to simulate the multi-writer registers in Aghazadeh et al.'s implementation from single-writer registers in constant time and constant space. This results in an implementation of an ℓ-bit single writer register from k-bit single writer registers which works for $k \in \Omega(\log n)$ and has $\Theta(\ell/k)$ step complexity. In this case, G only needs to contain $\Theta(n)$ buffers, so the space complexity is reduced to $\Theta(n\ell/k)$.

Now we turn our attention to Chaudhuri and Welch's regular register implementation. They use a complete binary tree where each leaf represents a different register value and each internal node stores a *switch*, a shared binary regular register that selects between its two children. Their regular register read operation traverses down the tree, following the switches, until it reaches a leaf and returns the value of that leaf. Their regular register write operation starts at the leaf with the value it wishes to write and traverses up the tree, changing each switch on its path to point towards that leaf.

Using Chaudhuri and Welch's implementation for the regular ℓ-bit registers in Vidyasankar's algorithm gives an implementation of an atomic ℓ-bit register from $\Theta(2^\ell)$ regular binary registers and one atomic binary register with $\Theta(\ell)$ step complexity. We call this the CWV implementation. The CWV implementation can be used in place of our tree

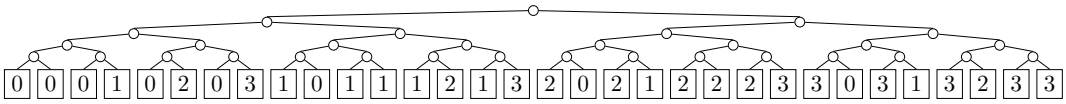

Figure 2 Atomic 4-valued register.

based implementation in our final optimal implementation. However, our implementation of a counter, which is based on our tree based implementation, is faster than the CWV implementation by a factor of 2. This counter will be used in our buffer based implementation.

4 Tree Based Implementation

We begin by showing how Chaudhuri and Welch's regular register implementation can be extended to an atomic register implementation. Then we show how both these implementations can be generalized to work for $k > 1$. In this section, it is more natural to talk about m-valued registers, where $m = 2^\ell$, rather than ℓ-bit registers.

4.1 Implementing an m-valued atomic register

Consider Chaudhuri and Welch's regular register implementation where each binary switch is atomic rather than regular. Notice that the WRITE operation in this implementation is atomic in the case where only one switch is changed on the path from the leaf to the root. This observation is the key behind our atomic register implementation. To construct an atomic register, we use a larger tree such that, for every pair of values, there are two leaves with these values that have a common parent. This allows us to change from the current value to any new value by changing only one switch. $\binom{m}{2}$ height 1 nodes are needed to guarantee this property, but we use m^2 height 1 nodes to simplify the implementation.

More formally, we construct a complete, perfectly balanced binary tree with m^2 height 1 nodes, $w_0, w_1, ..., w_{m^2-1}$. Each internal node stores a shared binary register called a switch which selects between its two children. All the binary registers can be regular except for those in the height 1 nodes, which must be atomic. Figure 2 illustrates an atomic 4-valued register.

The node w_i has a left child with value $\alpha = \lfloor i/m \rfloor$ and a right child with value $\beta = (i \bmod m)$. So each pair of values (α, β) has a common parent $w_{\alpha * m + \beta}$.

Algorithm 1 presents the pseudo-code for Chaudhuri and Welch's implementation, which we will use as a subroutine. A REGULARWRITE(i) operation changes the switches in the tree to point towards the leaf with index i. The fields of each node are immutable except for *switch*. The variable *root* and the contents of the array *leaves* are also immutable, and *node* is a local variable. Immutable variables and fields can be stored in the local memory of each process, instead of being stored in shared memory.

Our atomic READ algorithm starts at the root and returns the value of the leaf that it arrives at by following switches, just like REGULARREAD.

Our atomic WRITE(val) operation first computes the height 1 node, *parent*, whose left child has the current value and whose right child has the value being written. Then it performs REGULARWRITE with the index of its left child. Next, it changes *parent*'s switch to point to its right child. This step is exactly the same as performing a REGULARWRITE with the index of its right child, because all other switches on the path to the root remain the same. The WRITE operation is linearized immediately after this step.

Algorithm 1 Chaudhuri and Welch's implementation of a regular m-valued register.

```
0: procedure RegularRead()
1:     node ← root
2:     while node is not a leaf do
3:         s ← read(node.switch)
4:         if s = 0
5:         then node ← node.left
6:         else node ← node.right
7:     return node.value
```

```
0: procedure RegularWrite(index)
1:     node ← leaves[index]
2:     while node is not the root do
3:         if node is a left child
4:         then write(node.parent.switch, 0)
5:         else write(node.parent.switch, 1)
6:         node ← node.parent
```

Algorithm 2 Implementation of an atomic m-valued register.

```
0: procedure Read()
1:     return RegularRead()
```

```
0: procedure Write(val)
1:     parent ← parent(oldval, val)
2:     RegularWrite(parent.left.index)
3:     write(parent.switch, 1)
4:     oldval ← val
```

Pseudo-code for our atomic register implementation is presented in Algorithm 2. In the pseudo-code, *oldval* is a persistent local variable that is initialized to the initial value of the register. The function **parent**$(oldval, val)$ is performed locally and returns the height 1 node whose left child has value *oldval* and whose right child has value *val*. Both Read and Write operations take $\Theta(\log m)$ steps. This immediately implies that they are wait-free.

Fix an execution of Read and Write operations. The variables are initialized so that it appears as if a complete Write of 0 has occurred before any Read operation. Let R be a Read operation in this execution which returns α. If there is a Write of α linearized in the execution interval of R, then linearize R immediately after the linearization point of the first such Write operation. In this case, R returns the value of the last Write operation linearized before it.

Now suppose there is no Write of α linearized in the execution interval of R. In this case, we can linearize R immediately after its first step. We need to show that the last Write operation W linearized before the first step of R is a Write of α.

Since W is linearized immediately after its last step, W completes before R begins. If the writer does not start another Write until after R completes, then R is concurrent with no RegularWrite operations and, hence, returns the value written by W. So, suppose the writer starts another Write before R completes. Let W' be the first Write operation following W.

Suppose, for contradiction, that W returns $\beta \neq \alpha$. Since W' is linearized at line 3, line 3 of W' occurs after the first step of R by definition of W. Notice that a Write operation can be viewed as two RegularWrite operations. This is because the atomic **write** performed on line 3 is equivalent to RegularWrite(*parent.right.index*), since the writer performed RegularWrite(*parent.left.index*) immediately beforehand.

Therefore the last complete RegularWrite operation before the start of R is either the second RegularWrite of W or the first RegularWrite of W'. From the code, we can see that both operations write the value β.

R consists of a single RegularRead, so by the correctness of Algorithm 1, there exists a RegularWrite of value α concurrent with R. The first such RegularWrite operation must be the second RegularWrite of some Write operation W''. This is because the first RegularWrite of each Write operation writes the same value as the second

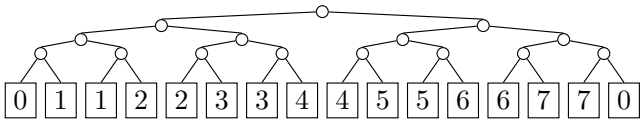

Figure 3 Atomic 8-valued counter.

RegularWrite of the previous Write operation. Since the second RegularWrite of W'' is atomic and W'' is linearized at the following configuration, W'' is linearized in the execution interval of R which contradicts our assumption. Therefore W must have written α.

▶ **Theorem 1.** *Algorithm 2 implements an atomic m-valued register.*

4.2 Implementing a Modulo m Counter

It is easy to modify our atomic m-valued register implementation to implement an atomic modulo m counter. Since the value can only be incremented, we only need one height 1 node for each $(\alpha, (\alpha+1) \bmod m)$ pair. Hence the space complexity can be reduced to $\Theta(m)$. ReadCounter performs the same steps as the atomic register Read, and Increment is the same as Write. Figure 3 illustrates an atomic 8-valued counter.

4.3 Extension to k-bit registers

To extend Chaudhuri and Welch's implementation and our atomic counter implementation to use k-bit registers, for $k > 1$, we replace the binary tree in each implementation with a 2^k-ary tree, and keep the same set of leaves. This simple modification reduces the height of the tree to $\Theta(\log_{2^k}(m))$, so that all operations have step complexity $\Theta((\log m)/k)$. We use one k-bit register for each internal node so the space complexity is equal to the number of internal nodes. Since there are $\Theta(m/2^k)$ height 1 nodes, there are $\Theta(m/2^k)$ internal nodes and the space complexity is also $\Theta(m/2^k)$.

For our atomic register implementation, we partition the values into sets of size 2^{k-1}, and have one height 1 node for each pair of these sets. This construction ensures every pair of values are siblings in the tree. Thus, we only need $(m/2^{k-1})^2$ height 1 nodes. Therefore, the step complexity becomes $\Theta((\log m)/k)$ and the space complexity becomes $\Theta(m^2/4^k)$. An m-valued register is the same as an ℓ-bit register when $m = 2^\ell$, so the step complexity is $\Theta(\ell/k)$ and the space complexity is $\Theta(4^{\ell-k})$. If $\ell \leq \lceil(\log_2 n)/2\rceil$, we have $m \leq 2\sqrt{n}$, which means the implementation has space complexity $O(n/4^k)$.

5 Buffer Based Implementation

We begin by describing the handshake object, a primitive that we use in our implementation. In Section 5.2, we describe our buffer based implementation and analyse its step and space complexities. Then, in Section 5.3, we introduce notation that is used in its correctness proof. Some important handshaking properties are proven in Section 5.4. The proof of correctness appears in Section 5.5.

5.1 Handshaking

A handshake object H is used to coordinate between pairs of processes and can be implemented using a pair of 1-bit single-writer registers $H.r$ and $H.w$. We will use a handshake object H_i

in the buffer based implementation to coordinate between the reader p_i and the writer. The idea of handshaking first appeared in papers by Peterson [9] and Lamport [7]. The version we will use is from Attiya and Welch [2].

We say that reader p_i *requests help* when it sets $H_i.r$ equal to the value it read from $H_i.w$. The writer *acknowledges a help request from* p_i when it sets the value of $H_i.w$ to be the opposite of what it read from $H_i.r$. The reader p_i *checks for an acknowledgement* by checking if these two handshake bits are different and the writer *checks for a help request from* p_i by checking if these two handshake bits are the same.

Intuitively, the handshake object guarantees that a check for an acknowledgement by p_i returns true if and only if the writer has acknowledged a help request from p_i since p_i's last help request. Similarly, a check for a help request from p_i by the writer returns true if and only if there has been a help request by p_i since the writer's last acknowledgement to p_i.

5.2 Description

As in Peterson's implementation, our buffer-based implementation uses an array of buffers G and a pointer V to the currently active buffer in G. However, G contains $4n$ buffers, instead of 2, and V is implemented using a single-incrementer modulo $4n$ counter, as described in Section 4.2. Like Aghazadeh, Golab and Woelfel's implementation, our implementation uses round-robin helping, except that it uses handshaking and completion bits to coordinate the helping.

In round-robing helping, each WRITE operation only helps a single reader. If the current WRITE operation helps process p_i, then the next WRITE operation will help process $p_{(i+1) \bmod n}$. This ensures that each reader p_i is helped once every n WRITE operations.

A reader begins by reading the counter V and announcing that value. This value is the index of the element in G that it tries to read. A WRITE operation that sees the announcement helps the reader by sending it the requested element of G, as well as the value that it just wrote. Reader p_i announces the value it read from V using an array A_i of size $\lceil (logn)/k \rceil$. It is the only process that writes to this array and only the writer reads from this array. The writer uses buffers M_i and N_i to send the value that was requested and the value it just wrote, respectively, to p_i. Only the writer can write to these two buffers, and only the reader p_i can read from these two buffers. The arrays A_i, M_i and N_i are accompanied by a completion bit which is set and cleared only by the process that writes to the array. The completion bit is set after a complete write to the array and while the completion bit is set, the array will not be written to. The completion bit of A_i is reset at the beginning of a READ$_i$ operation, and the completion bits of M_i and N_i are reset when the writer notices a new help request from p_i.

A WRITE(v) operation first computes the reader p_i that it should help and checks for a help request from that reader. If there is no help request from p_i, the writer writes v into buffer $G[(V+1) \bmod 4n]$ and increments V to point to this buffer. If there is a help request, the writer clears the completion bits of N_i and M_i, and acknowledges the help request. Next it writes v into buffers $G[(V+1) \bmod 4n]$ and N_i, increments V, and sets the completion bit of N_i. In either case, if the completion bit of M_i is not set and the completion bit of A_i is set, the writer copies the contents of buffer $G[A_i]$ into M_i and sets the completion bit of M_i.

A READ operation by reader p_i first clears the completion bit of its announcement array A_i and requests help. Then it announces the current value of V and sets the completion bit of A_i. If the writer has sent an acknowledgement and the completion bit of N_i is set, the reader returns the value in N_i. Otherwise, the reader reads buffer $G[A_i]$ and performs another check; if the writer still has not sent an acknowledgement or if the completion bit of

Algorithm 3 Buffer based implementation of an ℓ-bit register from k-bit registers.

$G[0 \dots 4n-1]$: array of buffers
V: modulo $4n$ counter
For each $i = 0, \dots, n-1$:
  N_i: buffer
  M_i: buffer
  F_{N_i}, F_{M_i}: completion bits
  $H_i.w$: writer's handshaking bit
  A_i: array of $\lceil (\log 4n)/k \rceil$ register
  F_{A_i}: completion bit
  $H_i.r$: reader's handshaking bit

```
0:  procedure READ_i()
1:      write(F_{A_i}, 0)
2:      t ← read(H_i.w)
3:      write(H_i.r, t)
4:      version ← READCOUNTER(V)
5:      writearray(A_i, version)
6:      write(F_{A_i}, 1)
7:      h ← H_i.r ≠ read(H_i.w)
8:      if h ∧ read(F_{N_i}) then
9:          val ← readarray(N_i)
10:     else
11:         val ← readarray(G[version])
12:         h ← H_i.r ≠ read(H_i.w)
13:         if h ∧ read(F_{M_i}) then
14:             val ← readarray(M_i)
15:     return val
```

```
0:  procedure WRITE(value)
1:      version ←
            (READCOUNTER(V) + 1) mod 4n
2:      i ← version mod n
3:      if read(H_i.r) = H_i.w then
4:          write(F_{N_i}, 0)
5:          write(F_{M_i}, 0)
6:          t ← read(H_i.r)
7:          write(H_i.w, 1 − t)
8:      writearray(G[version], value)
9:      INCREMENT(V)
10:     if ¬F_{N_i} then
11:         writearray(N_i, value)
12:         write(F_{N_i}, 1)
13:     if ¬F_{M_i} ∧ read(F_{A_i}) then
14:         rversion ← readarray(A_i)
15:         writearray(M_i, G[rversion])
16:         write(F_{M_i}, 1)
```

M_i is not set, the reader returns the value it read from $G[A_i]$. Otherwise, the writer has sent an acknowledgement and M_i contains the value that was requested, $G[A_i]$, so the reader reads and returns the value in M_i.

Pseudo-code for our implementation is presented in Algorithm 3. For two arrays of registers, v_1 and v_2, we use **writearray**(v_1, v_2) to denote copying the value of v_2 into v_1 one register at a time. **readarray**(v_1) denotes reading from v_1 one register at a time and returning the concatenation of the values that were read. These operations are not atomic operations. We will use the convention that 0 represents FALSE and 1 represents TRUE.

The step complexities of READ and WRITE are $\Theta(\ell/k + (\log n)/k)$ because reading and incrementing V takes $\Theta((\log n)/k)$ steps, copying a buffer takes $\Theta(\ell/k)$ steps, and copying an announcement array takes $\Theta((\log n)/k)$ steps. Since both operations have bounded step complexity, the implementation is wait-free.

A total of $6n$ buffers, a size n announcement array, $5n$ bits, and a modulo $4n$ counter are used, so the overall space complexity is $\Theta(n\ell/k + n(\log n)/k)$.

5.3 Notation

Here we define the notation that we will use throughout the proof.

If O is an operation, $C(O, c)$ refers to the configuration immediately after O completes line c of the code for that operation. If line c of O is a function call, then $C(O, c)$ is the

configuration immediately after the linearization point of the function call. If line c of O is a **writearray** operation then $C(O, c)$ indicates the configuration immediately after the completion of the entire operation.

READ$_i$ is a READ operation performed by process p_i. WRITE$_i$ is a WRITE operation that helps reader p_i. This means $i = (version \bmod n)$ on line 2 of a WRITE$_i$ operation.

5.4 Properties of Handshaking Bits

The following two properties about the handshaking bits, $H_i.r$ and $H_i.w$, follow immediately from properties proved by Attiya and Welch [2].

P1. Let C be a configuration such that $H_i.r \neq H_i.w$ and let R be the last READ$_i$ operation such that $C(R, 3) \to C$. Then there exists a WRITE$_i$ operation W such that $C(R, 2) \to C(W, 7) \to C$.

P2. Let C be a configuration such that $H_i.r = H_i.w$ and let R be the last READ$_i$ operation such that $C(R, 3) \to C$. Then there does not exist a WRITE$_i$ operation W such that $C(R, 3) \to C(W, 6) \to C(W, 7) \to C$.

In addition, the handshaking bits satisfy the following two properties.

P3. If $H_i.r \neq H_i.w$ at C and $H_i.r = H_i.w$ at a later configuration C', then there exists a READ$_i$ operation R such that $C \to C(R, 3) \to C'$.

P4 If $H_i.r = H_i.w$ at C and $H_i.r \neq H_i.w$ at a later configuration C', then there exists a WRITE$_i$ operation W such that $C \to C(W, 7) \to C'$.

Both properties have analogous proofs so we will only present the proof of **P3**.

Proof. Suppose, for contradiction, that **P3** is violated in some execution. Let C be the first configuration at which $H_i.r \neq H_i.w$ and there exists a later configuration C' at which $H_i.r = H_i.w$, but there is no READ$_i$ operation R such that $C \to C(R, 3) \to C'$. Consider the earliest such configuration C'. $H_i.r$ is not written to between C and C' because $H_i.r$ is only written to by line 3 of a READ$_i$ operation. Suppose, without loss of generality, that $H_i.r = 0$ and $H_i.w = 1$ at C. Then $H_i.r = H_i.w = 0$ at C' and there exists a WRITE$_i$ operation W that changes $H_i.w$ from 1 to 0 between C and C' by performing line 7. This means $H_i.r = 1$ at $C(W, 6)$, so $C(W, 6) \to C$. There must have been a READ$_i$ operation R that changed $H_i.r$ from 1 to 0 by performing line 3 between $C(W, 6)$ and C. Thus, $H_i.w = 0$ at $C(R, 2)$. Since $C \to C(W, 7)$ and $H_i.w = 1$ between $C(W, 6)$ and $C(W, 7)$, it follows that $C(R, 2) \to C(W, 6)$. At $C(R, 2)$, $H_i.r = 1$ and $H_i.w = 0$. At $C(W, 6)$, $H_i.r = 1$ and $H_i.w = 1$. Since $C(R, 2) \to C(W, 6) \to C(R, 3)$, there does not exist a READ$_i$ operation R' such that $C(R, 2) \to C(R', 3) \to C(W, 6)$. This contradicts the definition of C. Therefore **P3** holds. ◀

5.5 Proof of Correctness

Consider an arbitrary execution of READ and WRITE operations. Each WRITE operation W is linearized at $C(W, 9)$, which is immediately after it increments V.

Let R be a READ$_i$ operation by process p_i. If the check on line 8 of R returns FALSE, then R is linearized at $C(R, 4)$. We will prove that the value R returns is equal to the value of $G[V]$ at line 4 of R. If the check on line 8 of R returns TRUE, then R is linearized at $C(W, 12)$, where W is the last WRITE$_i$ operation such that $C(R, 2) \to C(W, 7) \to C(R, 7)$. In this case, we will prove that R returns the value written by W.

All local variables, in particular *version*, are initialized to 0. All global variables are also initialized to 0. This initial configuration can be viewed as the result of a complete WRITE(0) operation. Thus every READ operation is preceded by a complete WRITE operation.

Intuitively, the following lemma says that, if the test on line 8 of a READ_i operation R returns TRUE, then a WRITE_i operation has acknowledged the help request from R. This lemma also shows that the linearization point of R exists and is within its execution interval.

▶ **Lemma 2.** *Let R be a READ_i operation by process p_i. If $H_i.r \neq H_i.w$ at $C(R,7)$ and $F_{N_i} = 1$ at $C(R,8)$, then there exists a WRITE_i operation W such that $C(R,2) \to C(W,7) \to C(R,7)$. Furthermore, $C(W,12) \to C(R,8)$ for any such W.*

Proof. Suppose that $H_i.r \neq H_i.w$ at $C(R,7)$ and $F_{N_i} = 1$ at $C(R,8)$. By **P1**, there exists a write operation W such that $C(R,2) \to C(W,7) \to C(R,7)$. Let W be any such operation. Since line 4 of W sets F_{N_i} to 0 and $F_{N_i} = 1$ when p_i performs line 8 of R, F_{N_i} must have changed from 0 to 1 between $C(W,4)$ and $C(R,8)$. Therefore $C(W,12) \to C(R,8)$. ◀

Similarly, the following lemma says that if the test on line 13 of a READ_i operation R returns TRUE, then a WRITE_i operation has acknowledged the help request from R. Its proof can be obtained from the proof of the previous lemma by replacing $C(R,7)$ and $C(R,8)$ with $C(R,12)$ and $C(R,13)$.

▶ **Lemma 3.** *Let R be a READ_i operation by process p_i. If $H_i.r \neq H_i.w$ at $C(R,12)$ and $F_{N_i} = 1$ at $C(R,13)$, then there exists a WRITE_i operation W such that $C(R,2) \to C(W,7) \to C(R,12)$.*

Informally, the following lemma states that N_i will not change between any configuration where $H_i.r \neq H_i.w$ and $F_{N_i} = 1$, and the next execution of line 3 of a READ_i operation. This means that the reader can read safely from N_i on line 9.

▶ **Lemma 4.** *Let C be a configuration where $H_i.r \neq H_i.w$ and $F_{N_i} = 1$. Let R be the first READ_i operation such that $C \to C(R,3)$. Then N_i is not written to between C and $C(R,3)$.*

Proof. Suppose N_i was written to between C and $C(R,3)$. Then there must be a WRITE_i operation W such that some step of line 11 of W is executed between C and $C(R,3)$. From the code, $F_{N_i} = 0$ from $C(W,10)$ until W performs line 12. Since $F_{N_i} = 1$ at C, it follows that $C \to C(W,10)$ and F_{N_i} changed from 1 to 0 between C and $C(W,10)$. So line 4 of some write operation W' was performed in this interval. It follows from the code that $H_i.r = H_i.w$ at $C(W',3)$.

Suppose C occurred between $C(W',3)$ and $C(W',4)$. By **P4**, line 7 of some WRITE operation must have occurred between $C(W',3)$ and C, which is impossible since there is only one writer. Therefore C occurred before $C(W',3)$.

In this case, $H_i.r$ and $H_i.w$ changed from being unequal to being equal between C and $C(W',3)$. By **P3**, line 3 of some READ_i operation was performed between C and $C(W',3)$, which means it was performed between C and $C(R,3)$. This contradicts the choice of R. Therefore N_i cannot be written to between C and $C(R,3)$. ◀

The following lemma is an analogous statement for M_i. It guarantees that the reader can read safely from M_i on line 14. Its proof can be obtained from the proof of Lemma 4 by changing each occurrence of N_i to M_i and changing line numbers appropriately.

▶ **Lemma 5.** *Let C be a configuration where $H_i.r \neq H_i.w$ and $F_{M_i} = 1$. Let R be the first READ_i operation such that $C \to C(R,3)$. Then M_i is not written to between C and $C(R,3)$.*

For the remainder of this section, R will represent an arbitrary READ_i operation. We will show that R returns the input to the last WRITE operation linearized before R. Let v denote the value of V that process p_i reads on line 4 of R, and let g be the ℓ-bit value in $G[v]$ at configuration $C(R, 4)$. It follows from the next lemma that R must return g when R is linearized at $C(R, 4)$.

▶ **Lemma 6.** *g is the input to the last* WRITE *operation linearized before $C(R, 4)$.*

Proof. Let W be the last WRITE operation to be linearized before $C(R, 4)$. Suppose that g was not the input to W. Since W writes to $G[v]$ before it is linearized, another WRITE operation must have written to $G[v]$ between the linearization point of W and $C(R, 4)$. After W is linearized, V must be incremented $4n - 1$ times before *version* is reassigned value v on line 1 and $G[v]$ is written to on line 8. Thus, at least $4n - 1$ other WRITE operations are linearized after W and before $C(R, 4)$. This contradicts the choice of W. ◀

Lemma 7 is used to show that a complete WRITE_i operation between $C(R, 4)$ and $C(R, 7)$ will cause the check on line 8 of R to return TRUE. It is also used to prove Lemma 8.

▶ **Lemma 7.** *If there is at least one complete WRITE_i operation W between $C(R, 4)$ and the end of R, then $H_i.r \neq H_i.w$ and $F_{N_i} = 1$ from the end of W to the end of R.*

Proof. By lines 3–7 and 10–12 of the code for WRITE_i, it follows that $H_i.r \neq H_i.w$ and $F_{N_i} = 1$ at the end of W. No READ_i operation performs line 3 between $C(R, 4)$ and the end of R, so, by the contrapositive of **P3**, $H_i.r \neq H_i.w$ from the end of W until the end of R. This also implies that line 4 of the code for WRITE_i is not performed from the end of W until the end of R, so F_{N_i} remains equal to 1 until the end of R. ◀

Suppose there are two complete WRITE_i operations between $C(R, 4)$ and the end of R. Lemma 8 implies that no write to M_i occurs between the end of the second WRITE_i operation and the end of R. This will later be used to show that R reads g from M_i if it executes line 14.

▶ **Lemma 8.** *If the test on line 8 of R evaluates to* FALSE *and there are at least two complete WRITE_i operations between $C(R, 4)$ and the end of R, then $F_{M_i} = 1$ from the end of the second WRITE_i operation to the end of R.*

Proof. Let W and W' be the first and second WRITE_i operations between $C(R, 4)$ and the end of R. Suppose, for contradiction, that W finishes before $C(R, 7)$. Then, by Lemma 7, $H_i.r \neq H_i.w$ at $C(R, 7)$ and $F_{N_i} = 1$ at $C(R, 8)$, so the test on line 8 of R will evaluate to TRUE. This contradicts the assumption that line 8 of R evaluates to FALSE. Therefore W finishes after $C(R, 7)$ so W' starts after $C(R, 7)$.

From the code, we see that $F_{A_i} = 1$ from $C(R, 7)$ to the end of R, so $F_{A_i} = 1$ during the execution of W'. It follows from lines 13–16 of the code for WRITE_i that $F_{M_i} = 1$ at the end of W'. By Lemma 7, $H_i.r \neq H_i.w$ between the end of W and the end of R. Therefore line 5 of the code for WRITE_i is not performed from the end of W' until the end of R. Hence F_{M_i} remains equal to 1 from the end of W' until the end of R. ◀

Intuitively, the following lemma captures the idea that, as long as $H_i.r = H_i.w$ or $F_{M_i} = 0$, $G[v]$ is equal to g. This lemma shows that R must have read g on line 11 if the check on line 13 returns FALSE.

▶ **Lemma 9.** *If R read a value other than g from $G[v]$ on line 11, then $H_i.r \neq H_i.w$ at $C(R, 12)$ and $F_{M_i} = 1$ at $C(R, 13)$.*

Proof. Suppose that R read a value other than g from $G[v]$ on line 11. Note that, between $C(R,4)$ and $C(R,11)$, line 3 of READ_i is not performed, so the value of $H_i.r$ does not change.

By definition of g, a WRITE operation must have written to $G[v]$ between $C(R,4)$ and $C(R,11)$. Since V has value v at $C(R,4)$, V must be incremented $4n-1$ times after $C(R,4)$ before *version* is reassigned value v on line 1 of the WRITE and $G[v]$ is written to on line 8. This implies there are at least two full WRITE_i operations W' and W'' that helped process p_i between $C(R,4)$ and $C(R,11)$. Suppose W'' occurs after W'. By Lemma 7, $H_i.r \neq H_i.w$ and $F_{N_i} = 1$ from the end of W' until the end of R. Since R performed lines 11–13, the test on line 8 evaluated to FALSE. Therefore by Lemma 8, $F_{M_i} = 1$ from the end of W'' until the end of R. Since W'' finished before $C(R,11)$, it follows that $H_i.r \neq H_i.w$ at $C(R,12)$ and $F_{M_i} = 1$ at $C(R,13)$. ◀

The proof that R reads g on line 14 if the check on line 13 returns TRUE is presented below.

▶ **Lemma 10.** *If $H_i.r \neq H_i.w$ at $C(R,12)$ and $F_{M_i} = 1$ at $C(R,13)$, then R read g from M_i on line 14.*

Proof. Suppose $H_i.r \neq H_i.w$ at $C(R,12)$ and $F_{M_i} = 1$ at $C(R,13)$. By Lemma 3, there exists at least one write operation W such that $C(R,2) \to C(W,7) \to C(R,12)$. Let W be the last such write operation. Since W sets F_{M_i} to 0 on line 5 and $F_{M_i} = 1$ at $C(R,13)$, there exists a write operation W' (possibly equal to W) such that $C(W,5) \to C(W',16) \to C(R,13)$. Since there is only one writer, W' is either equal to W or starts after W finishes. Hence $C(W,7) \to C(W',13)$.

$F_{A_i} = 1$ at $C(W',13)$ because W' executed line 16. From the code, we see that $F_{A_i} = 0$ at $C(R,1)$ and only line 6 of a READ_i operation sets F_{A_i} to 1. Thus $C(R,6) \to C(W',13)$.

Next, we prove that *rversion* equals v at $C(W',14)$. Since $C(R,6) \to C(W',14) \to C(W',16) \to C(R,13)$, a complete execution of line 14 of W' occurs after R writes v to A_i on line 5 and before $C(R,13)$. From the code, we see that $A_i = v$ from $C(R,5)$ until the end of R, so $rversion = v$ at $C(W',14)$. Since *rversion* is a local variable, $rversion = v$ while W' performs line 15.

Suppose, for contradiction, that $G[v]$ was written to between $C(R,4)$ and $C(W',15)$. The writer writes to $G[v]$ before incrementing V to have value v on line 9. Since V has value v at $C(R,4)$, it follows that, between $C(R,4)$ and $C(W',15)$, V must have been incremented at least $4n-1$ times. Thus there are at least two complete WRITE_i operations between these two configurations. Since R performed lines 11–13, the test on line 8 evaluated to FALSE. By Lemma 8, $F_{M_i} = 1$ from the end of the second WRITE_i operation, W'', to the end of R. Since $C(R,4) \to C(W'',1) \to C(W',15) \to C(R,13)$, $F_{M_i} = 1$ at $C(W',15)$. This is impossible, since W' performs line 15 only if $F_{M_i} = 0$ at $C(W',13)$ and does not change the value of F_{M_i} until line 16. Therefore $G[v]$ was not written to between $C(R,4)$ and $C(W',15)$.

Be definition, $G[v] = g$ at $C(R,4)$, so $G[v] = g$ from $C(R,4)$ until $C(W',15)$. In particular, $G[v] = g$ while W' executed line 15. Hence $M_i = g$ at $C(W',15)$.

We claim that $H_i.r \neq H_i.w$ from $C(W,7)$ to the end of R. By the choice of W, we see that line 7 of a WRITE_i operation does not occur between $C(W,7)$ and $C(R,7)$. Since $H_i.r \neq H_i.w$ at $C(R,7)$, we know that $H_i.r \neq H_i.w$ from $C(W,7)$ to $C(R,7)$ by **P4**. From the code for READ_i, we see that line 3 of a READ_i operation is not executed between $C(R,7)$ and the end of R. Therefore by **P3**, $H_i.r \neq H_i.w$ from $C(R,7)$ to the end of R. Since $C(W,7) \to C(W',16) \to C(R,13)$, it follows that $H_i.r \neq H_i.w$ at $C(W',16)$.

From the code, we can see that $F_{M_i} = 1$ at $C(W',16)$ and M_i does not change between $C(W',15)$ and $C(W',16)$. Since $C(R,4) \to C(W',16)$, the first READ_i operation after

$C(W', 16)$ occurs after the end of R. So, by Lemma 5, M_i remains equal to g from $C(W', 16)$ to the end of R. In particular, $M_i = g$ at $C(R, 14)$. ◀

Finally, we show that R returns the input to the last WRITE operation linearized before R. First suppose that, $H_i.r \neq H_i.w$ at $C(R, 7)$ and $F_{N_i} = 1$ at $C(R, 8)$. By Lemma 2, there is a WRITE$_i$ operation W such that $C(R, 2) \to C(W, 7) \to C(R, 7)$. Let W be the last such WRITE$_i$ operation. Also by Lemma 2, line 12 of W is executed and $C(W, 12) \to C(R, 8)$. R is linearized at $C(W, 12)$ and W is linearized at $C(W, 9)$, so W is the last WRITE operation linearized before R. Let α be the input to W. Note that N_i equals α at $C(W, 12)$.

We claim that $H_i.r \neq H_i.w$ from $C(W, 7)$ to the end of R. By the choice of W, we see that line 7 of a WRITE$_i$ operation does not occur between $C(W, 7)$ and $C(R, 7)$. Since $H_i.r \neq H_i.w$ at $C(R, 7)$, we know that $H_i.r \neq H_i.w$ from $C(W, 7)$ to $C(R, 7)$ by **P4**. From the code for READ$_i$, we see that line 3 of a READ$_i$ operation is not executed between $C(R, 7)$ and the end of R. Therefore by **P3**, $H_i.r \neq H_i.w$ from $C(R, 7)$ to the end of R.

In particular, since $C(W, 7) \to C(W, 12) \to C(R, 8)$, $H_i.r \neq H_i.w$ at $C(W, 12)$. When W executes line 12, it sets F_{N_i} to 1. It follows from Lemma 4 that $N_i = \alpha$ from $C(W, 12)$ to line 3 of the next READ$_i$ operation after R, so N_i equals α at $C(R, 9)$. Thus, R returns the value written by W, as required.

Now suppose $H_i.r = H_i.w$ at $C(R, 7)$ or $F_{N_i} = 0$ at $C(R, 8)$. In this case, R is linearized at $C(R, 4)$. By Lemma 6, g is the value written by the last WRITE operation linearized before $C(R, 4)$. Thus, it suffices to prove that R returns g. Consider two subcases depending on the result of the test on line 13 of R. If the test returns false, then by the contrapositive of Lemma 9, the value that R read from $G[rversion]$ on line 11 is equal to g. If the test returns true, then by Lemma 10, R read g from M_i on line 14. So in either case, R returns g.

▶ **Theorem 11.** *Algorithm 3 implements an atomic ℓ-bit register.*

6 Step Lower Bound of Register Implementation

So far, we have presented implementations of an atomic ℓ-bit n-reader single-writer register from atomic k-bit n-reader single-writer registers. For the lower bound, we consider the case where the large register only needs to be regular and there is only one reader. This results in a stronger lower bound.

▶ **Theorem 12.** *Any regular ℓ-bit single-reader, single-writer register implementation from atomic k-bit single-writer registers with $O(\ell/k)$ step complexity for* READ *requires $\Omega(\ell/k)$ step complexity for* WRITE.

Proof. Let $t = \lfloor \ell/k \rfloor$. Since READ operations have step complexity $O(t)$, there are positive integers α and t_0 such that, for all $t \geq t_0$, each READ operation performs at most αt steps. Let $t \geq t_0$. We consider executions starting from an initial configuration I in which the writer completes its first WRITE, crashes, and then a single READ occurs. Let u be the number of steps performed by the writer in the worst case. If $u > \alpha t$, the lower bound holds, so suppose that $u \leq \alpha t$. The READ algorithm can be represented by a decision tree of height at most αt, where each node represents the read of a shared k-bit register. Without loss of generality, we may assume that no shared k-bit register is read more than once on any root to leaf path.

For each of the 2^ℓ different values w that can be written by the writer, consider the path in this decision tree taken by the reader. Let $E(w) = \{(i, v) \mid$ if the shared k-bit register read at depth i contains value $v \neq$ its value in $I\}$. The set $E(w)$ uniquely specifies the leaf that the reader reaches and, thus, is an encoding of w. Since $1 \leq i \leq u$ and there are $2^k - 1$

choices for each value v (i.e. excluding the value of the k-bit register in configuration I), the number of different possible encodings is

$$\sum_{j=0}^{u}\binom{\alpha t}{j}(2^k-1)^j \le \sum_{j=0}^{u}\binom{\alpha t}{u}\binom{u}{j}(2^k-1)^j = \binom{\alpha t}{u}\sum_{j=0}^{u}\binom{u}{j}(2^k-1)^j = \binom{\alpha t}{u}(2^k)^u$$
$$\le (2^k)^u(\alpha te/u)^u = (\alpha e\cdot 2^k t/u)^u.$$

Thus $2^\ell \le (\alpha e\cdot 2^k t/u)^u$. Taking the logarithm of both sides yields $\ell \le u(\log_2(\alpha e)+k+\log_2(t/u))$. Since $k\le \ell/t$ and $t<\ell$, it follows that $\ell \le u(\log_2(\alpha e)+\ell/t+\log_2(t/u))$ and $1\le (u/t)\cdot(\log_2(\alpha e)t/\ell+1+\log_2(t/u)t/\ell) < (u/t)\cdot(\log_2(\alpha e)+1+\log_2(t/u))$. Setting $\beta = t/u$ gives $1\le(\log_2(\alpha e)+1+\log_2\beta)/\beta$. Thus $\beta-\log_2\beta\le\log_2(\alpha e)+1\in O(1)$. This implies that $t/u=\beta\in O(1)$, because $\lim_{\beta\to\infty}\beta-\log_2\beta=\infty$. Therefore $u\in\Omega(t)=\Omega(\ell/k)$. ◀

7 Conclusion

We presented two new implementations of large ℓ-bit single-writer registers from small k-bit single-writer registers, which work for all $k\ge 1$. We can combine them as follows: use the first implementation if $\ell\le\lceil(\log_2 n)/2\rceil$; otherwise, use the second implementation. This results in an implementation of large single-writer registers with optimal step complexity and $\Theta(n\ell/k)$ space complexity.

We proved that any implementation with $O(\ell/k)$ step complexity for Read requires $\Omega(\ell/k)$ step complexity for Write. Since Read of an ℓ-bit register requires at least $\lceil\ell/k\rceil$ **reads** of k-bit registers, our lower bound shows that our implementation is step optimal.

It would be interesting to find a more space efficient implementation with optimal step complexity or to prove a lower bound on the amount of space required. We have some algorithms with $o(n\ell/k)$ space complexity, but slightly worse step complexity, so there may be a trade-off between space complexity and step complexity.

It would also be interesting to see if ℓ-bit multi-writer registers can be implemented from k-bit multi-writer registers with $\Theta(\ell/k)$ step complexity for any $k\ge 1$. Unfortunately, we do not know of any way to modify our tree based implementation to obtain a multi-writer regular (or atomic) register. Without an efficient counter that supports multiple incrementers, we can not extend our buffer based implementation to obtain a multi-writer register, either.

Acknowledgements. We would like to thank our supervisor, Professor Faith Ellen, for the many insightful discussions that we've had together, and for the numerous hours she put into editing our work.

References

1 Zahra Aghazadeh, Wojciech Golab, and Philipp Woelfel. Making objects writable. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, pages 385–395. ACM, 2014.

2 Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. John Wiley Interscience, March 2004.

3 Soma Chaudhuri, Martha J. Kosa, and Jennifer L. Welch. One-write algorithms for multi-valued regular and atomic registers. *Acta Inf.*, 37(3):161–192, 2000.

4 Soma Chaudhuri and Jennifer L. Welch. Bounds on the costs of multivalued register implementations. *SIAM J. Comput.*, 23(2):335–354, 1994.

5 M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

6 A. Israeli and A. Shaham. Time and space optimal implementations of atomic multi-writer register. *Information and Computation*, 200(1):62–106, 2005.

7 Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–101, 1986.

8 Andreas Larsson, Anders Gidenstam, Phuong Hoai Ha, Marina Papatriantafilou, and Philippas Tsigas. Multiword atomic read/write registers on multiprocessor systems. *ACM Journal of Experimental Algorithmics*, 13, 2008.

9 G. L. Peterson. Concurrent reading while writing. *ACM Trans. Program. Lang. Syst.*, 5(1):46–55, 1983.

10 K Vidyasankar. A very simple construction of 1-writer multireader multivalued atomic variable. *Information Processing Letters*, 37(6):323–326, 1991.