One-Write Algorithms for Multivalued Regular and Atomic Registers[†]

Soma Chaudhuri[‡] Martha J. Kosa[§] Jennifer L. Welch[¶]

June 1, 2001

Abstract

This paper presents an algorithm for implementing a k-valued regular register (the *logical* register) using $k(k-1)/2$ binary regular registers (the *physical* registers) that requires only one physical write per logical write. The same algorithm using binary atomic registers implements a k-valued atomic register. The algorithm is simple to describe and depends on properties of paths in a related graph. Two lower bounds are given on the number of registers required by one-write implementations in the regular case. The first lower bound, $2k - 1 - \lfloor \log k \rfloor$, holds for a fairly general class of algorithms. The second lower bound holds for a restricted class of implementations and implies that our algorithm is optimal for this class. Both lower bounds improve on the best previously known lower bound, which was k. The two lower bounds also hold for the atomic case under further restrictions.

Key Words: concurrent reading and writing, shared registers, regular, atomic, complexity analysis, lower bounds.

1 Introduction

In any concurrent system, processes need to communicate with other processes. Concurrent reads and writes of shared memory cells, or registers, are required for communication. A consistency condition specifies what guarantees are provided concerning the values returned in the presence of concurrent accesses. Lamport

[†]This work was supported in part by NSF grant CCR-9010730, an IBM Faculty Development Award, and NSF Presidential Young Investigator Award CCR-9158478. The work of the first author was also supported in part by NSF grant CCR-9308103, and the work of the second author was also supported in part by a UNC Board of Governors Fellowship. Much of this work was performed while the authors were with the Department of Computer Science, University of North Carolina at Chapel Hill.

[‡]Current address: Department of Computer Science, Iowa State University, Ames, IA 50011

[§]Current address: Department of Computer Science, Tennessee Technological University, Cookeville, TN 38505. Contact Author. E-mail: mjk9504@tntech.edu

[¶]Current address: Department of Computer Science, Texas A&M University, College Station, TX 77843-3112

introduced three conditions, in increasing order of strength, safe, regular and atomic [6], thus defining three types of registers. If the shared memory provides a stronger guarantee, then it is often easier for users to program the system, but implementing the shared memory may be more difficult. Thus it is helpful to know which types of registers can implement which other types and what the costs of these implementations are. Many such implementations have been developed; [5] surveys many representative algorithms.

In this paper we focus on implementing a k-ary regular (respectively, atomic) register, the *logical* register, out of binary regular (respectively, atomic) registers, the *physical* registers, for $k > 2$. A *register* is a memory cell that supports concurrent reading and writing by a collection of processes; we assume there are several readers but only one writer. A *k-ary* register can take on k different values; *binary* means 2-ary. A read of a *regular* register must return either the value of the most recent preceding write (a well-defined notion since there is only one writer) or the value of an overlapping write. For an *atomic* register, reads must behave like reads of regular registers; additionally, for any two nonoverlapping reads, the value returned by the read completed first must not have been written for the first time after the value returned by the second read. [1]

More specifically, we are interested in *one-write algorithms*—implementations with the property that every WRITE to the logical register requires only one write to a physical register and no reads of physical registers.[2] Since bounds on the number of physical accesses per logical access can be converted into time bounds for the logical access, a one-write algorithm would have time-efficient logical WRITEs, perhaps an important characteristic for applications in which WRITEs outnumber READs.

In this paper, we present a one-write algorithm for implementing a k-ary regular register out of binary regular registers. Clearly this algorithm is optimal in the number of physical writes per logical WRITE. The best previous upper bound was $\lceil \log k \rceil$ writes per WRITE, due to Chaudhuri and Welch [2]. The algorithm is simple to describe using the complete graph whose nodes are labeled with the logical values. Its correctness proof is based on properties of paths in this graph. The same algorithm also implements a k-ary atomic register out of binary atomic registers.

One drawback of our algorithm is that it requires $C(k,2) = k(k-1)/2 = O(k^2)$ physical registers. The best previous lower bound on the number of physical registers for a k-ary regular implementation was k [2], for any number of physical writes per logical WRITE. Thus binary to k-ary regular implementations are inherently expensive in the amount of "hardware" required. In this paper we show two improved lower bounds on the number of physical registers in any one-write algorithm in which the writer does not read physical registers. Each lower bound holds for a natural class of regular implementations. The first lower bound, $2k-1-\lfloor \log k \rfloor$, holds for a reasonably unrestrictive class of implementations satisfying the *symmetric* property: Suppose at some point the current value of the logical register is v and a logical WRITE for w changes physical register x. Then if the next logical WRITE is for v, x is changed again (back to its previous

[1] The weakest guarantee, safety, only ensures that a read which does not overlap a write returns the value of the latest preceding write; the value returned by a read that does overlap a write can be arbitrary. [2] studied one-write implementations of k-ary safe registers out of binary safe registers and showed that $\Theta(k)$ physical registers are necessary and sufficient.

[2] Logical operation names will be in upper case and physical operation names will be in lower case throughout this paper.

value). The second lower bound holds for a restricted class of implementations satisfying the *toggle* property: for each unordered pair of logical values, there is one particular register that is changed when the logical register switches between those values. This lower bound implies that our algorithm is optimal in the number of physical registers for this class.

We developed a general transformation to convert a one-write algorithm for k values into a one-write algorithm for $k-1$ values using fewer physical registers. This transformation is used in the inductive proof of our symmetric lower bound. In proving these two lower bounds, we have developed considerable understanding of such one-write algorithms. We prove that, for any one-write algorithm (in which the writer does not read physical registers), there is no advantage, in terms of number of physical registers, to be gained if readers write, or if different readers follow different protocols, or if a reader's protocol depends on its history. Furthermore, for symmetric algorithms, there is no advantage if a reader reads some physical registers more than once. Thus our lower bound proofs are simpler, since we assume the reader does none of the above. These results are shown with transformation techniques similar to the one mentioned previously.

Our lower bounds also apply for the two corresponding classes of atomic implementations which prohibit readers from doing any of the above. However, in the atomic case, the restrictions on the readers are probably unreasonably strong.

In Section 2, we present our basic definitions. Section 3 contains the algorithm and in Section 4 we prove it is correct with respect to regularity and atomicity. Section 5 consists of our lower bounds. We conclude in Section 6.

Some of the results of this paper have appeared in preliminary form in [1].

2 Definitions

2.1 Wait-Free Register Implementations

We use a simplified form of the I/O automaton model [7] to describe our system.

To implement a logical register with value set V, where $|V| = k$, we compose a collection of physical registers X_j, $1 \leq j \leq m$, each with value set $\{0,1\}$, a collection of read processes RP_i, $1 \leq i \leq n$, and a single write process WP. The read and write processes implement the protocols used by the readers and writer of the logical register. Each such protocol consists of accessing certain of the physical registers and doing some local computation.

Communication between these components takes place via **actions**. Each action is an output of one component (the component that generates it) and an input to another component. Components are modeled as state machines in which actions trigger transitions. Components have no control over when inputs occur, and thus must have a transition for every input in every state. Components do have control over when outputs occur; if an output labels a transition from a state, then the output is **enabled** in that state.

An **execution** of the implementation consists of a sequence in which state tuples (one entry for the state of each component) and actions alternate, beginning with a tuple of initial states. For each action π in the execution, π must be enabled in the preceding state of the component for which it is an output. In the following state tuple, the states of the two components for which π is an input and an output must change according to the transition functions, while the remaining components' states are unchanged.

A **schedule** is the sequence of actions in an execution.

The **logical actions** are READ(i), RETURN(i, v), WRITE(v), and ACK, $1 \leq i \leq n$ and $v \in V$. READ(i) is an input to RP_i from the outside world and RETURN(i, v) is an output from RP_i to the outside world. WRITE(v) is an input to WP from the outside world and ACK is an output from WP to the outside world. Although we do not explicitly model the outside world with a component, we do assume that for each i, the outside world and RP_i cooperate so that READs and RETURNs strictly alternate, beginning with a READ, and analogously for WP.

The **physical actions** are $\mathrm{read}_j(i)$, $\mathrm{return}_j(i, v)$, $\mathrm{write}_j(v)$, and ack_j. The subscript j is between 1 and m; it indicates that X_j is the physical register being read or written. The parameter v is either 0 or 1 and indicates the value being read from or written to X_j. The parameter i is between 0 and n and indicates which of the read and write processes is reading X_j ($i = 0$ indicates the write process). For a fixed j, there is no parameter i for writes and acks, since there is a unique read or write process that writes X_j.

A READ(i) and its following RETURN(i, v) form a **logical operation**, as do a WRITE(v) and its following ACK. **Physical operations** are defined analogously. An operation is **pending** if its first half is present but not its second half; if both halves are present, it is **completed**.

We assume that the read and write processes cooperate with the physical registers so that for each i, $0 \leq i \leq n$, and each j, $1 \leq j \leq m$, $\mathrm{read}_j(i)$ and $\mathrm{return}_j(i, *)$ alternate beginning with a read, and analogously for writes. We also assume that no read or write process has a physical operation pending unless it has a logical operation pending.

The read and write processes must work together to implement a logical register in a "wait-free" manner. Informally, an implementation is wait-free if any logical operation initiated by a process can complete in a finite number of steps regardless of the actions of the other processes in the system. However, the wait-free property involves fairness considerations because a process cannot complete a logical operation if it is not allowed to take steps in its protocol. An execution is **fair** to a process if every physical operation initiated by the process eventually completes and if no output action of the process is continuously enabled without occurring. We formally define an implementation to be **wait-free** if for any fixed process, in any execution which is fair to that process, every logical invocation by that process has a matching response. Our algorithms actually provide a bounded number of actions, while our lower bounds hold for algorithms satisfying the weaker definition.

2.2 Regular and Atomic Registers

A physical register is **regular** if, in every execution, it satisfies:

- **Regular Property.** Every completed physical read operation returns a value written by an overlapping write operation or by the most recent preceding write (or the initial value if there is no preceding write).

A physical register is **atomic** if, in every execution e, it satisfies:

- **Atomic Property.** There exists a **linearization** [3] of all the completed physical operations and some subset of the pending physical write operations in the execution, i.e., there is a permutation T of these operations[3] in e such that (1) the ordering of non-overlapping operations in T is the same as their ordering in e (two operations do not overlap if the response for one occurs before the call of the other one), and (2) each read in T returns the value written by the latest preceding write in T (or the initial value if there is no preceding write).

  An equivalent definition of atomicity, which we will use when convenient, is that there exists a point somewhere between the invocation and response of each operation, called its **linearization point**, such that ordering the operations according to their linearization points produces a linearization. Informally, the linearization point is when the operation takes effect.

Regular and atomic *logical* registers are defined analogously, replacing physical operations with logical operations.

2.3 One-Write Algorithms

To describe a register implementation algorithm, it is sufficient to describe the code for the read and write processes. An algorithm is a **one-write algorithm** if, in every execution, every logical WRITE uses at most one physical write and no physical reads.

We now define several terms which will be used in the discussion of one-write algorithms.

Let A be a one-write algorithm that uses m binary registers. A **configuration** of A is an element C of $\{0,1\}^m$; let $C[i]$ denote the i^{th} bit of C for $i \in \{1,\ldots,m\}$. The **distance** between two configurations C_1 and C_2, denoted $d(C_1,C_2)$, is the number of bits that differ in C_1 and C_2. Configurations C_1 and C_2 are **neighbors** if $d(C_1,C_2)=1$. A configuration C is **initial** if $C[i]$ is the value of the i^{th} binary register in the initial state of A for all $i \in \{1,\ldots,m\}$. A configuration C is **reachable** if there exists a state in an execution of A where no physical write is pending such that $C[i]$ is the value of the i^{th} binary register in the state for all $i \in \{1,\ldots,m\}$. (If a physical write is pending, the value of that physical register is ambiguous.)

[3] For purposes of the permutation T, a single entity $\text{read}(j,i,v)$ to represent a read operation is created from a completed read operation in e consisting of the matching, but separated, actions $\text{read}_j(i)$ and $\text{return}_j(i,v)$; similarly, a single entity $\text{write}(j,v)$ is created from a pending or completed write operation in e starting with the action $\text{write}_j(v)$.

3 The Algorithm

In this section, we present our one-write algorithm.

Let V be the value set of the logical register, where $|V| = k$ and $v_0 \in V$ is the initial value. Without loss of generality, we can assume that $V = \{1, \ldots, k\}$ and $v_0 = 1$. Let K_V be the complete graph with k nodes and $r = C(k, 2)$ edges in which each edge is labeled with a distinct number from the set $\{1, \ldots, r\}$ and each node is labeled with a distinct element from V. The **special bit set** corresponding to $v \in V$ is defined as $s(v) = \{l | l$ labels an edge incident to the node labeled v in $K_V\}$. Since K_V is a complete graph, $|s(v)| = k - 1$ for all $v \in V$.

Our algorithm uses r binary regular registers (bits). Each bit corresponds to an edge of K_V. A reader reads all r bits and returns the value of a function f applied to the configuration obtained. The function f is defined below. The writer changes a bit only when the value of the logical register changes; when the value is changed from v to w, the bit whose label is contained in $s(v) \cap s(w)$ is changed. There is exactly one such bit because there is exactly one edge connecting v and w in K_V. Figure 1 is a formal description of our algorithm.

We now define f. For each $v \in V$ and configuration C, let $count(C, v) = \Sigma_{i \in s(v)} C[i]$. If $\{v | count(C, v)$ is odd$\}$ is empty, then let $f(C) = 1$. Otherwise, let $f(C) = \max\{v | count(C, v)$ is odd$\}$. Since any configuration C induces a subgraph of K_V consisting of all nodes and all edges corresponding to bits whose values are 1 in C, there are an even number of nodes in this subgraph with odd degree by a basic result in graph theory. These nodes correspond to the values with odd counts in C, implying that the cardinality of $\{v | count(C, v)$ is odd$\}$ is even.

To explain and analyze the algorithm, it is helpful to partition configurations into valid and invalid. Configuration C is **valid** if either (1) $count(C, v)$ is even for all $v \in V$, or (2) $count(C, 1)$ is odd and $count(C, w)$ is odd for exactly one $w \neq 1$. Otherwise, C is invalid.

If a configuration C is valid, then there is a path in K_V, not necessarily edge-disjoint, starting from the node labeled with $v_0 = 1$ and corresponding to initial configuration 0^r such that when the path is traversed and the bit labeling an edge is changed when the edge is traversed, then the resulting configuration is C. The intermediate nodes in the path correspond to the sequence of logical values written in the execution fragment leading to configuration C. The resulting node is labeled v, where $v = f(C)$. For each $i \in \{1, \ldots, r\}$, $C[i]$ is the parity of the number of times edge i is traversed in this path. Suppose the path corresponding to valid configuration C does not end at the node labeled with 1. The two endpoints of the path are adjacent to an odd number of edges in the path, while all internal nodes are adjacent to an even number. The last node in the path is entered one more time than it is left; thus, the count for that node is odd. The first node in the path is left one more time than it is entered; thus, the count for that node is odd. All other nodes are entered and left the same number of times; thus, the counts for those nodes are even. C satisfies condition (2) of the definition of valid. Suppose the path corresponding to valid configuration C is cyclic. All nodes

Physical Registers (Bits): $X_1, \ldots, X_r$, initially $X_j = 0$, for all $j \in \{1, \ldots, r\}$

Reader i, $1 \leq i \leq n$: local variables $x_1, \ldots, x_r$

READ(i):

for $j := 1$ to r do

read$_j(i)$

return$_j(i, x_j)$

endfor

RETURN($i, f(x_1 \ldots x_r)$)

Writer: local variables $x_1, \ldots, x_r$, initially $x_j = 0$, for all $j \in \{1, \ldots, r\}$, and old, initially $old = 1$

WRITE(v):

if $v \neq old$ then

pick the unique i from $s(v) \cap s(old)$

write$_i(\overline{x_i})$

ack$_i$

$x_i := \overline{x_i}$

$old := v$

endif

ACK

Figure 1: One-Write Algorithm

in the cycle are adjacent to an even number of edges in the cycle. All nodes in the cycle are entered and left the same number of times; thus, the counts for all the nodes are even. C satisfies condition (1) of the definition of valid.

The focus in this paper is on the costs of accessing shared memory and not on the costs of local computation. The writer performs at most one shared memory access per WRITE. Each reader performs $O(k^2)$ shared memory accesses per READ. In addition, each reader must compute f for each configuration that it observes. However, computing f adds no more to the asymptotic time complexity of the work performed by each reader. READs are very expensive compared to WRITEs, but WRITEs are extremely time-efficient, which may be important for applications where WRITEs outnumber READs. However, Chaudhuri and Welch [2] proved that READs are inherently not cheap, by showing that at least k shared memory accesses are required by any one-write algorithm.

4 Proofs of Correctness

We first prove that our algorithm implements a k-ary regular register from binary regular registers in Subsection 4.1. We then prove in Subsection 4.2 that our algorithm implements a k-ary atomic register from binary atomic registers.

4.1 Proof of Regularity

Lemma 4.1 shows that any reachable configuration is valid and is mapped by f to the value which was written to the register by the last completed WRITE.

Lemma 4.1 *Let C be a reachable configuration resulting from a sequence of m physical writes corresponding to the logical values $v_1, v_2, \ldots, v_m$. Then C is valid, and $f(C) = v_m$.*

Proof We proceed by induction on m.

Basis: $m = 0$. Then C is the initial configuration and is valid, and $f(C) = 1$.

Inductive step: $m > 0$. Suppose the lemma is true for $m-1$. Now we show that it is true for m. Suppose the sequence of logical values is $v_1, v_2, \ldots, v_{m-1}, v_m$ and the sequence of corresponding reachable configurations is $C_1, C_2, \ldots, C_{m-1}, C_m$. By the inductive hypothesis, C_{m-1} is valid, and $f(C_{m-1}) = v_{m-1}$. If $v_{m-1} = v_m$, then clearly the lemma is true. Thus, suppose that $v_{m-1} \neq v_m$. There are two possibilities for v_{m-1}.

Case 1: $v_{m-1} = 1$. Then $count(C_{m-1}, v)$ is even for all $v \in V$. When the write for v_m is performed, the unique bit $b \in s(1) \cap s(v_m)$ is changed. Thus $count(C_m, 1)$ and $count(C_m, v_m)$ become odd, and $count(C_m, v)$ remains even for all $v \in V - \{1, v_m\}$. Therefore C_m is valid, and $f(C_m) = v_m$.

Case 2: $v_{m-1} \neq 1$. Then $count(C_{m-1}, 1)$ and $count(C_{m-1}, v_{m-1})$ are odd, and $count(C_{m-1}, v)$ is even for all $v \in V - \{1, v_{m-1}\}$. When the write for v_m is performed, the unique bit $b \in s(v_{m-1}) \cap s(v_m)$ is changed. There are two possibilities for v_m. First suppose that $v_m = 1$. Thus $count(C_m, 1)$ and $count(C_m, v_{m-1})$ become even, and $count(C_m, v)$ remains even for all $v \in V - \{1, v_{m-1}\}$. Therefore C_m is valid, and $f(C_m) = 1$. Otherwise suppose that $v_m \neq 1$. Thus $count(C_m, v_m)$ becomes odd, $count(C_m, 1)$ remains odd, and $count(C_m, v)$ is even for all $v \in V - \{1, v_m\}$. Therefore C_m is valid, and $f(C_m) = v_m$. ■

If a reader RETURNs value v, we must show that v was actually written to the register by some WRITE overlapping the READ or by the last WRITE preceding the READ. This is nontrivial because a slow reader can read either a reachable or an unreachable configuration by noticing traces from many WRITEs to the logical register by a fast writer. Lemma 4.2 shows that a WRITE(v) operation has occurred during or just before an interval in an execution if a bit in $s(v)$ is changed during that interval. Lemma 4.3 shows that if two valid configurations agree in all bits of $s(v)$ for some v and one is mapped to v by f, then the other must be mapped to v by f. Lemma 4.4 shows that an invalid configuration C agrees with some valid configuration

in the special bits corresponding to $f(C)$. Lemma 4.5, which shows that the reader will RETURN a correct value of the register no matter what configuration it reads, is the main result of this section. The proof of Lemma 4.5 uses Lemma 4.2 initially to deduce that if a value is not written to the logical register, then its special bit set remains unchanged. If the reader reads a reachable configuration, then Lemma 4.3 is applied to deduce the correctness of the value RETURNed. Otherwise, Lemmas 4.4 and 4.3 are applied to deduce the correctness of the value RETURNed.

Lemma 4.2 *For any interval in any execution, if no WRITE(v) operation overlaps the interval or occurs as the last preceding WRITE, then the bits in $s(v)$ are not changed during the interval.*

Proof Suppose in contradiction that a bit in $s(v)$ is changed during the interval. Then the value in the register changed from some w to v or the value in the register changed from v to some w. This is impossible because no WRITE(v) operation overlapped the interval or occurred as the last preceding WRITE. Therefore, the lemma is true. ■

Lemma 4.3 *Choose any valid configurations C and D. If $f(D) = v$ and $C[i] = D[i]$ for all $i \in s(v)$, then $f(C) = v$.*

Proof There are two cases.

Case 1: $v = 1$. Thus $count(D, w)$ is even for all $w \in V$. Since $C[i] = D[i]$ for all $i \in s(1)$, $count(C, 1) = count(D, 1)$. Thus $count(C, w)$ is even for all $w \in V$ because C is valid. This implies that $f(C) = 1$.

Case 2: $v \neq 1$. Thus $count(D, v)$ is odd. Since $C[i] = D[i]$ for all $i \in s(v)$, $count(C, v) = count(D, v)$; therefore, $count(C, v)$ is odd. Thus $count(C, 1)$ is odd and $count(C, w)$ is even for all $w \in V - \{1, v\}$ because C is valid. This implies that $f(C) = v$. ■

Lemma 4.4 *Choose any invalid configuration C. Let $f(C) = v$. Then there exists a valid configuration D such that $C[i] = D[i]$ for all $i \in s(v)$ and $f(D) = v$.*

Proof Let V_{odd} be the set of all values w in V such that $count(C, w)$ is odd. Recall that $|V_{odd}|$ is even. Since C is invalid, V_{odd} is not empty, so V_{odd} contains at least two values.

Case 1: $1 \in V_{odd}$. Then $V_{odd} = \{1, v_1, \ldots, v_{2h-1}\}$ for some h, where $1 < v_1 < \ldots < v_{2h-1}$. Thus, $f(C) = v_{2h-1}$. Let D be the result of starting with C and flipping the $h - 1$ bits corresponding to the edges $(v_1, v_2), (v_3, v_4), \ldots, (v_{2h-3}, v_{2h-2})$ in K_V.

The result of flipping the bit corresponding to (v_i, v_{i+1}) is to change the counts of v_i and v_{i+1} from odd to even, while leaving all other counts the same. Thus in D, only 1 and v_{2h-1} have odd counts. Clearly D is valid. By the definition of f, $f(D) = v_{2h-1}$.

Since $f(C) = v = v_{2h-1}$, the construction guarantees that $C[i] = D[i]$ for all $i \in s(v)$ and $f(C) = f(D)$.

Case 2: $1 \notin V_{odd}$. Then $V_{odd} = \{v_1, v_2, \ldots, v_{2h}\}$ for some h, where $1 < v_1 < v_2 < \ldots < v_{2h}$. Thus, $f(C) = v_{2h}$. Let D be the result of starting with C and flipping the h bits corresponding to the edges $(1, v_1), (v_2, v_3), \ldots, (v_{2h-2}, v_{2h-1})$ in K_V. Thus in D, only 1 and v_{2h} have odd counts. Clearly D is valid. By the definition of f, $f(D) = v_{2h}$.

Since $f(C) = v = v_{2h}$, the construction guarantees that $C[i] = D[i]$ for all $i \in s(v)$ and $f(C) = f(D)$. ■

Lemma 4.5 *Let C be the configuration obtained by a reader during some execution of the READ protocol. Suppose $f(C) = v$. Then the value v was written by a WRITE which overlapped the READ or the value v was the result of the last WRITE preceding the READ.*

Proof Assume for contradiction that the value v was not written by a WRITE which overlapped the READ and the value v was not the result of the last WRITE preceding the READ. Thus no state of the algorithm during the READ has the physical registers in a configuration with value v. By Lemma 4.2, the bits in $s(v)$ are never changed during the READ. Let D be any reachable configuration resulting from either the last preceding WRITE or any overlapping WRITE. D is valid by Lemma 4.1, and $D[i] = C[i]$ for all $i \in s(v)$. There are two cases.

Case 1: Suppose C is valid. Since D has the same values as C for the bits in $s(v)$ and $f(C) = v$, $f(D) = v$ by Lemma 4.3, which is a contradiction.

Case 2: Suppose C is invalid. By Lemma 4.4, there exists a valid configuration C' such that $C[i] = C'[i]$ for all $i \in s(v)$ and $f(C') = v$. Thus $C'[i] = D[i]$ for all $i \in s(v)$. By Lemma 4.3, $f(D) = v$, which is a contradiction. ■

The logical register is regular by Lemma 4.5. It is seen to be wait-free by inspecting the code of the read and write processes. Thus we have:

Theorem 4.6 *A one-write algorithm for implementing a k-ary regular register from binary regular registers exists.*

4.2 Proof of Atomicity

We now assume that the constituent binary registers are atomic. To show that the logical register is atomic, we must construct a linearization of the logical operations for an arbitrary fair execution e. The WRITEs will be linearized in the order in which they occur (remember that there is only one writer of the logical register). To define the placement of the READs, we consider them in the order in which they end, yielding a total ordering of the READs, denoted $R_1, R_2, \ldots$. For each READ, R_i, we show the existence of a WRITE that will be considered the WRITE from which R_i reads. Then we will show how to use this WRITE to

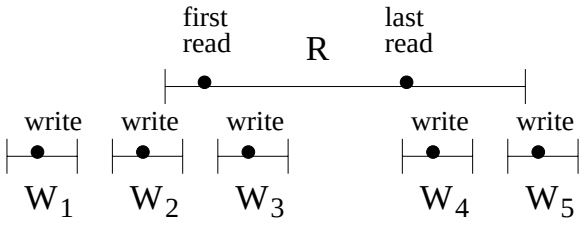

Figure 2: An example of the definition of PRS; solid circles are linearization points of physical operations.

place R_i appropriately. The WRITE for R_i must write the same value that R_i reads. Furthermore, the choices of the WRITEs for all the READs must be consistent so that if READ R_j precedes READ R_i, then the WRITE for R_i does not precede the WRITE for R_j.

Lemma 4.8 shows that each READ has at least one WRITE that is a potentially correct choice. Lemma 4.9 shows that the WRITEs can be chosen from the set of potential choices in such a way as to avoid the precedence inversion just mentioned.

We define several terms which will be used in proving that our algorithm satisfies the atomic property.

Since the physical registers are atomic, there exists a linearization L of the physical operations in the execution e. Fix such an L. Define the **physical linearization point** of WRITE W (with respect to L) to be the linearization point of the physical write inside W if there is one; otherwise define it to be the RETURN of W.

The **possible writes-to-read set** for READ R, denoted PRS(R), is the set of all WRITES W whose physical linearization points are between (with respect to L) the first and last physical reads of R. In addition, PRS(R) contains the WRITE whose physical linearization point immediately precedes the first physical read in R. Without loss of generality, we assume that each execution has a special initializing WRITE, containing a physical write, which precedes all other operations in the execution. Thus, PRS(R) is always nonempty.

Note that PRS(R) is a consecutive sequence of WRITEs in the execution e. Furthermore, PRS(R) is a, possibly proper, subset of the set of all WRITEs that overlap or immediately precede R. An example of this situation is given in Figure 2, in which $\{W_1, \ldots, W_5\}$ is the set of WRITEs that immediately precede or overlap R, while PRS(R) = $\{W_2, W_3\}$. In general, a prefix and a suffix of the sequence of WRITEs that immediately precede or overlap a READ R might be missing from PRS(R). Therefore, we can say that PRS(R) is the set of WRITEs that overlap or immediately precede R *in a more restricted sense*: it only includes the WRITEs whose physical linearization points are between physical reads of R, or immediately precedes a physical read of R.

The next lemma shows an important relationship between the PRS sets of two non-overlapping READs.

Lemma 4.7 *If READ R finishes before READ R' begins, then the last WRITE of PRS(R) either is the first WRITE of PRS(R') or precedes the first WRITE of PRS(R').*

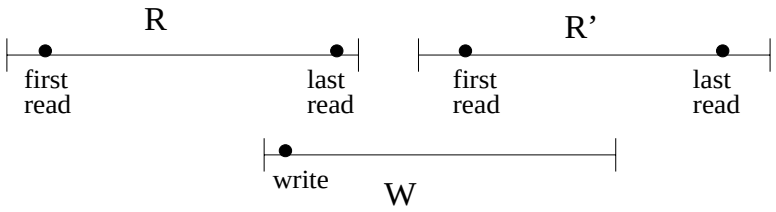

Figure 3: Situation in Proof of Lemma 4.7; solid circles are linearization points of physical operations.

Proof Consider the last WRITE W of PRS(R). The physical linearization point for W must precede the last read in R and therefore must precede the first read in R'. If there is no WRITE whose physical linearization point is between W's physical linearization point and the first read in R', then W is the first WRITE of PRS(R'), as shown in Figure 3. Otherwise, W precedes the first WRITE of PRS(R'). ■

If W is a logical WRITE, then **value**(W) = v if the call for W is WRITE(v). If R is a logical READ, then **value**(R) = v if the response for R is RETURN(i, v) for some i. The **same value set** for READ R, denoted by SVS(R), is $\{W \in \text{PRS}(R) | \text{ value}(W) = \text{value}(R)\}$. A useful property of SVS(R) is described in the following lemma.

Lemma 4.8 *SVS(R) is nonempty, for every READ R.*

Proof To prove that SVS(R) is non-empty, it is sufficient to prove that there exists a WRITE in PRS(R) whose value is value(R). Lemma 4.5 states that there exists a WRITE among the WRITEs that immediately precede or overlap R whose value is value(R). But since PRS(R) might be a proper subset of the set of WRITEs that immediately precede or overlap R, we cannot immediately conclude that W is in PRS(R).

However, a careful look shows that the proof for Lemma 4.5 and the proofs of its supporting lemmas hold with the more restricted definition of overlap that we use in defining PRS. Therefore, Lemma 4.5 proves that there exists a WRITE W in PRS(R) whose value is value(R). This implies that SVS(R) is non-empty. ■

The next lemma shows that we can choose an appropriate WRITE in SVS(R_i) to avoid precedence inversions.

Lemma 4.9 *There exists a function π from the READs in e to the WRITEs in e such that for all $i \geq 1$, the following are true.*

1. *$\pi(R_i)$ is in SVS(R_i).*
2. *For all $j < i$, if R_j precedes R_i in e, then $\pi(R_i)$ does not precede $\pi(R_j)$ in e.*

Proof We define π by induction on i, the index of the READs.

Basis: $i = 1$. Lemma 4.8 implies that SVS(R_1) is not empty. Choose any WRITE in SVS(R_1) as $\pi(R_1)$. Condition 2 is vacuously true since R_1 is the first READ to end.

Inductive step: $i > 1$. Assume π has been defined for $R_1, \ldots, R_{i-1}$. Again, Lemma 4.8 states that SVS(R_i) is not empty.

Consider all READs R_j such that R_j precedes R_i. Thus $j < i$ and $\pi(R_j) = W_j$ has already been defined for all such j. Let W' be the latest WRITE among all the W_j's and choose k such that $\pi(R_k) = W'$. We must show that there is some W in SVS(R_i) that does not precede W' (and thus does not precede any W_j's) and thus can be chosen as $\pi(R_i)$.

Lemma 4.7 shows that the last WRITE of PRS(R_k) is or precedes the first WRITE of PRS(R_i). Since $W' \in$ PRS(R_k) and SVS(R_i) $\subseteq$ PRS(R_i), W' is either equal to or precedes the first WRITE in SVS(R_i). Therefore, no WRITE in SVS(R_i) can precede W' and the feared precedence inversion cannot occur. ■

We can use Lemma 4.9 to determine where to place each READ in our proposed linearization. As usual, we consider the READs in the order in which they end. Define Σ_i inductively as follows. Let Σ_0 be the sequence of WRITEs in e, in order. For $i > 0$, let Σ_i be obtained from Σ_{i-1} by placing R_i immediately before the first WRITE following $\pi(R_i)$. (If no WRITE follows $\pi(R_i)$, then place R_i at the end of Σ_{i-1}.)

If e contains a finite number of READs, say m, then let $T = \Sigma_m$. Otherwise, let $T = \lim_{i \to \infty} \Sigma_i$.

To see that the linearization T is well-defined, we need to check that there is only a finite number of READs linearized before any given WRITE. Suppose not. Then there is a pair of consecutive WRITEs W_1 and W_2 in T such that $W_1 = \pi(R_i)$ for an infinite number of READs R_i. In e, each of these READs must start before W_2 ends, implying that in e there is an infinite number of READs which start within a finite time. This is impossible since we have a finite number of readers and e is a sequence.

The following theorem proves that our algorithm satisfies the atomic property. Its proof follows simply from the construction of the linearization.

Theorem 4.10 *T is a linearization of e.*

5 Lower Bounds on Number of Registers

We have proven the existence of a one-write algorithm for implementing a k-ary regular (respectively, atomic) register from binary regular (respectively, atomic) registers. The number of registers used by our algorithm is very large, $C(k, 2) = O(k^2)$. The best previously known lower bound on the number of registers for this problem is k, shown by Chaudhuri and Welch [2].

In this section we show two improved lower bounds for the problem, under certain restrictions. We primarily consider regular registers; the implications of these results for atomic registers are discussed in Section 5.5.

We begin, in Section 5.1, by proving that any one-write algorithm (in the regular case) can be converted to a convenient normal form. Section 5.2 presents the key definitions and lemma needed in our lower bounds. Our main result, in Section 5.3, is a lower bound of $2k - 1 - \lfloor \log k \rfloor$ on the number of registers required by a class of algorithms we call "symmetric." The restriction to symmetric algorithms allows us to strengthen some of the definitions and to assume, without loss of generality, that readers do not read the same binary register more than once during a READ. An even more restrictive class of algorithms is defined in Section 5.4; our algorithm is shown to be tight for this class.

In three places in this section (Theorem 5.1, Theorem 5.3, and Lemma 5.5), we discuss how to convert one algorithm into another by replacing some of the accesses to physical registers with accesses to local variables in the read or write processes' local states. We also describe how to map an execution of the new algorithm back to an execution of the original algorithm; this mapping requires being able to identify accesses to local state variables during transitions, in order to convert them to accesses to physical registers. To tie the use of local state variables to the formal model of Section 2, imagine that the state of a process consists of a set of local variables and each state transition is described using an imperative programming language to manipulate the local variables. Such a convention is consistent with the description of the algorithm in Section 3. When an action occurs at a point in an execution, it triggers an application of the process' transition function based on the current state of the process. Then the code describing the transition function is executed, which accesses the local variables.

5.1 Conversion to Normal Form

An algorithm with the following properties is a **normal form** algorithm:

1. no reader performs a physical write,
2. every reader has the same program, and
3. every reader starts in the same state at the beginning of every READ.

In this subsection, we show how any regular one-write algorithm can be converted to a normal form regular one-write algorithm without increasing the number of physical registers. Assuming normal form algorithms simplifies our lower bound proofs.

Theorem 5.1 *Any regular one-write algorithm A using m physical registers can be converted to a normal form regular one-write algorithm A' which uses at most m physical registers.*

Proof Each reader's protocol in algorithm A' is the same as reader 1's protocol in algorithm A, starting in reader 1's initial state, except that the readers in algorithm A' do not perform any physical writes. We handle the missing physical writes in the following way. Consider the set of physical registers M_R that are

written to by the readers in algorithm A. Each reader contains a local variable c_i corresponding to each physical register i in M_R, and initialized with the same value. Now, whenever a reader accesses one of these registers i in algorithm A, it accesses the corresponding local bit c_i in algorithm A' instead. The writer's protocol in algorithm A' is the same as the writer's protocol in algorithm A. Since the writer does no physical reads in algorithm A, it never accesses the registers in M_R.

A' is therefore a normal form one-write algorithm. We now prove the regularity of A'. Consider any execution e' of algorithm A'. Let s' be the schedule of e'. We consider each completed READ r_i in e' in turn and show that it RETURNs a value that satisfies regularity. For r_i, we build a sequence of actions, s_i, which will be shown to be a schedule of a possible execution of algorithm A. We obtain s_i from s' by removing all READs other than r_i along with their associated physical actions (the reason for doing so is given below). We also change r_i to be a READ by process 1. We now consider each action a_{ij} within r_i in turn. First, a_{ij} itself is placed in s_i. Then, for each read or write to a local variable c_l that occurs in e' when a_{ij} executes, we place the corresponding physical actions (invocation and response) to physical register l after a_{ij}.

We now argue that there is an execution e_i of algorithm A with schedule s_i. Since the set of physical registers M_R are only accessed by the single READ r_i (the writer never accesses them), they basically function as local variables in s_i (this would not be true in the presence of multiple readers). Also, the fact that read process 1 is in its initial state at the beginning of r_i is consistent with algorithm A since this is its first READ (this would not be true if there were multiple READs). Therefore, schedule s' of algorithm A' translates to a valid schedule s_i of algorithm A.

Suppose r_i in e' RETURNs value v. Then the corresponding READ in execution e_i of algorithm A also RETURNs value v. By the assumption about A, e_i satisfies the regular property. Thus, in e_i, v is the value of an overlapping WRITE, the value of the last preceding WRITE, or the initial value of A (also, of A'). It follows that v is a proper value for r_i to RETURN in e' because the sequences of WRITEs in e_i and e' are the same, and e_i's READ and r_i have the same relationship with the WRITEs. Also, the fact that e_i does not include the actions of the other READs in e' is irrelevant since regularity (unlike atomicity) is not affected by the relationship between READs. Thus, r_i RETURNs a correct value. Since all completed READs in e' are regular and e' is an arbitrary execution, A' satisfies the regular property. ■

5.2 Constructibility

If normal form one-write algorithm A uses m binary registers, A has 2^m configurations. These configurations are nodes in a directed m-dimensional hypercube H_A. If configurations C_1 and C_2 are neighbors, then both (C_1, C_2) and (C_2, C_1) are edges of H_A. An edge (C_1, C_2) of H_A is an **algorithm edge** if C_1 is a reachable configuration and C_2 can be derived from C_1 after one WRITE operation. An edge (C_1, C_2) of H_A is labeled with i, where i is the bit in which C_1 and C_2 differ.

In our lower bound proofs, we want to deduce the value which must be RETURNed by a reader given a particular configuration. This mapping from configurations to values is given by a "value extraction

function," such as the function f from our algorithm in Section 3.

We now define the **value extraction function** $f_A : \{0,1\}^m \to V$ for an arbitrary normal form one-write algorithm A that implements a k-ary regular register from m binary regular registers. We first define bit i to be **consistent** with configuration C if the value of bit i is $C[i]$. $f_A(C)$ is defined to be the value RETURNed by a reader according to A if all the bits that the reader reads are consistent with configuration C and if the reader never sees two different values for the same bit during the READ. If no reader ever reads bits consistent with configuration C, then $f_A(C)$ is undefined. Thus f_A is a partial function.

We now discuss why f_A is well-defined. Consider two logical READs. Suppose the reader performing the first logical READ reads a subset S_1 of the physical registers, RETURNing v_1, and the reader performing the second logical READ reads a different subset S_2 of the physical registers, RETURNing v_2, where $v_1 \neq v_2$. Suppose all bits in $S_1 \cup S_2$ are consistent with C. This is impossible because the readers have the same program and start their READs in the same initial state. For the readers to read two different sets of physical registers, there must be some physical register for which the first reader obtained 1 and the second reader obtained 0 (or vice versa). Thus one of the readers did not read bits consistent with configuration C. Therefore, f_A is well-defined.

We now define terms which will be used in the formalization of our general technique for "fooling the reader," which is Lemma 5.2 below. Let A be a normal form one-write algorithm for implementing a k-ary regular register from m binary regular registers. Let S be a set of reachable configurations and C be a configuration. C is **constructible** from S if for each $i \in \{1, \ldots, m\}$, there exists a $C' \in S$ such that $C'[i] = C[i]$. (A similar definition was given in [4].) Let $f_A(S) = \{f_A(C) | C \in S\}$. S is **strongly connected** if for all distinct $D, E \in S$, there exists a directed path from D to E in H_A consisting only of algorithm edges in which every configuration on the path is an element of S.

Given a configuration C which is constructible from a strongly connected set of configurations S, Lemma 5.2 states that $f_A(C)$ must be in $f_A(S)$; otherwise, the reader could be fooled into returning a wrong value. In our lower bound proofs, we obtain a contradiction to Lemma 5.2 by identifying a strongly connected set S of configurations and showing how there is a constructible C with a wrong value.

Lemma 5.2 *Let A be a normal form one-write algorithm. For every configuration C and every strongly connected set S of reachable configurations, if C is constructible from S, then $f_A(C) \in f_A(S)$.*

Proof Suppose in contradiction that $f_A(C) \notin f_A(S)$. Consider the following execution of A. First the writer executes a sequence of WRITEs so that the resulting configuration is in S. This sequence exists because every configuration in S is reachable. Then a logical READ starts. For all i, whenever the reader is about to read bit i, the writer executes a sequence of WRITEs with the following properties: (1) the configuration after each WRITE is in S, and (2) the final configuration D is such that $C[i] = D[i]$. Since S is strongly connected, this sequence exists. Thus the reader returns $f_A(C)$, which violates the regular property because $f_A(C)$ was not the value of any overlapping WRITE or of the preceding WRITE. ■

5.3 Symmetric Property

A normal form one-write algorithm A has the **symmetric property** if, for all configurations C_1 and C_2 that are neighbors, (C_1, C_2) is an algorithm edge of H_A if and only if (C_2, C_1) is an algorithm edge of H_A. In other words, suppose the writer enters configuration C_1 due to writing logical value v and then goes to configuration C_2 due to writing logical value w. If the next logical value to be written is v again, then the writer must return to configuration C_1.

If A satisfies the symmetric property, the two directed edges connecting any pair of neighboring configurations are either both algorithm edges or both non-algorithm edges. Thus the two directed edges can be replaced by one edge which is either an algorithm edge or a non-algorithm edge. Therefore, H_A can be considered an undirected graph.

As a result, the statement of Lemma 5.2 can be simplified to refer to *connected sets* instead of strongly connected sets.

We begin by showing how an arbitrary symmetric algorithm can be transformed into a symmetric algorithm using no more registers in which every reader reads each physical register at most once during a READ. Thus we can assume without loss of generality that in a symmetric algorithm every reader reads each physical register at most once during a READ.

5.3.1 Conversion to No Repeated Reads

Theorem 5.3 *Any regular one-write symmetric algorithm A using m physical registers can be converted to a regular one-write symmetric algorithm A' using at most m physical registers in which every reader reads each physical register at most once.*

Proof Since the transformation of Theorem 5.1 does not change the writer's protocol, it follows that the transformation preserves symmetry. Therefore, we can assume that A is a normal form one-write symmetric algorithm.

We show how the writer can force the reader to always read the same value from a given physical register during a READ without affecting the values of other physical registers which may be read later. Whenever the reader is about to reread a physical register, the writer retraces its steps backwards to the latest configuration in which the physical register had the value returned by the original read. The symmetric property makes this possible. After the repeated read, the writer retraces its steps forwards to the configuration just before the repeated read.

The writer's protocol in A' is the same as the writer's protocol in A. The same set of physical registers M used in A is used in A'. The local state of the reader's protocol in A' contains local bits c_i and a_i corresponding to each physical register i in M. Local variable c_i, which holds a copy of physical register i, is initialized with the same value as physical register i; local variable a_i, which is a flag indicating if physical

register i has been accessed yet during the current READ, is initialized with 0. The reader's protocol in A' is the same as the reader's protocol in A, with the following exception. The reader in A' reads a_i before reading physical register i. If a_i contains 0, then the reader reads physical register i, copies the value read into c_i, and writes 1 to a_i. Otherwise, the reader reads c_i.

We now prove the regularity of A'. Consider any execution e' of algorithm A'. We consider each READ in e' separately and show that the value it returns is correct, according to the definition of regularity. Since the definition of a regular register is not concerned with the relationship between values returned by different READs, this simplification is possible.

Choose any read R_i in e' and let j be the process performing the READ. We will construct an execution e_i of the original algorithm A from execution e' as follows.

Let s' be the schedule of e'. Let s_i be the result of removing from s' all actions except those of the writer and those belonging to R_i. We now consider each action a that is part of R_i. (The possibilities for a are the invocation and response for R_i itself and for the reads of the physical registers contained within it.) We will insert a series of actions after a in s_i. For each read of a local bit c_l that occurs in e' during the execution of a, we do the following in order:

1. Let b be the value read from c_l.
2. Place actions in s_i to complete the pending WRITE W (if it exists).
3. Let C_t be the latest configuration in e' preceding a such that $C_t[l] = b$. In other words, physical register l has the same value in C_t that was read from local bit c_l, which is the value returned by the first (and only) read by process j from register l within the current READ in e'.
4. Let ws be the sequence of values written to the logical register since C_t (not including the pending WRITE W from step 2).
5. Place in s_i logical WRITEs (along with the associated physical writes) for the values in ws, in reverse order. By the symmetric property, after those WRITEs, the configuration of the physical registers will be C_t.
6. Place in s_i the actions $\text{read}_l(j)$ and $\text{return}_l(j, b)$.
7. Place in s_i logical WRITEs (along with the associated physical writes) for the values in ws, in order.
8. If there was a pending WRITE W in step 2, then place in s_i a repeat of the actions of W preceding action a. As a result, another copy of W will now be pending and the configuration of the physical registers will be the same as at the beginning of a.

By induction, there exists some execution e_i of A with schedule s_i.

Let v be the value RETURNed by READ R_i. Note that this is the same in e' and in e_i. By assumption on A, e_i satisfies the regular property and thus v is the value of a WRITE that either immediately precedes or overlaps R_i in e_i. By the way e_i was constructed from e', every WRITE that immediately precedes or overlaps R_i in e_i has the value of some WRITE that immediately precedes or overlaps R_i in e', since every such WRITE in e_i either is some WRITE in e' or is a repetition of some WRITE in e'. That is, while there may be additional WRITEs in e_i, *no additional logical values are written in* e_i. Thus, v is also a correct value to be returned by R_i in e'.

Since the above argument holds for every READ in every execution, A' satisfies the regular property.

■

5.3.2 Main Result

Choose any $k \geq 4$. (See Section 5.4 for a discussion of the case when $k = 3$.)

Let SYM(k) be the set of all one-write algorithms which implement a k-ary regular register from binary regular registers and satisfy the symmetric property.

Let $R(k)$ be the minimum number of binary registers required by any algorithm in SYM(k). The main result of this section is Theorem 5.10, which states that $R(k) > 2k - 2 - \lfloor \log k \rfloor$. Theorems 5.1 and 5.3 imply that we can assume that every algorithm in SYM(k) is normal form and readers do not read twice, since no algorithm without those properties can use fewer registers.

The proof of Theorem 5.10 is inductive. Lemma 5.4, which shows that 4 binary regular registers cannot implement a 4-ary regular register, forms the base case for the proof. In the inductive step, either k is a power of 2, or k is not a power of 2. If k is a power of 2, then Lemma 5.6, which proves that $R(k) \geq R(k-1)+1$, is used. If k is not a power of 2, then Lemma 5.7, which proves that $R(k) \geq R(k-1)+2$, is used. Then some algebraic manipulations enable us to derive the desired lower bound. The proofs of Lemmas 5.6 and 5.7 use Lemma 5.5, which gives conditions under which a one-write algorithm can be converted into a one-write algorithm for fewer logical values using fewer physical registers. The proof of Lemma 5.5 consists of a general algorithm transformation. The notation R_A indicates the number of binary registers used by algorithm A.

Lemma 5.4 $R(4) > 4$.

Proof Suppose in contradiction that there exists an algorithm A in SYM(k) such that $R_A = 4$. Suppose without loss of generality that $V = \{R, G, B, Y\}$, the initial configuration is 0000, $f_A(0000) = R$, $f_A(1000) = G$, $f_A(0100) = B$, and $f_A(0010) = Y$. The configuration 0000 must have neighbors with the three other colors because A is a one-write algorithm. We now attempt to assign values to the remaining 12 configurations.

Figure 4 shows the current assignment of values to configurations and the possibilities for some currently unassigned configurations. Because A is a one-write algorithm, we only need to consider configurations which differ in one bit from the last assigned configuration 1000. We cannot assign two different values to the same configuration. Thus, we have six choices to consider:

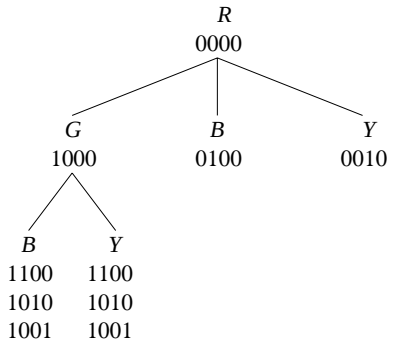

Figure 4: First Set of Choices in the Proof of Lemma 5.4

1. $f_A(1010) = B$ and $f_A(1100) = Y$.
2. $f_A(1010) = B$ and $f_A(1001) = Y$.
3. $f_A(1001) = B$ and $f_A(1100) = Y$.
4. $f_A(1100) = B$ and $f_A(1010) = Y$.
5. $f_A(1100) = B$ and $f_A(1001) = Y$.
6. $f_A(1001) = B$ and $f_A(1010) = Y$.

We can eliminate choices 1 and 2 by showing that $f_A(1010) \neq B$. $f_A(1010) \neq B$ because otherwise 0010 is constructible from the connected set $\{0000, 1000, 1010\}$ and $f_A(0010) = Y$ is not in $f_A(\{0000, 1000, 1010\}) = \{R, G, B\}$, contradicting Lemma 5.2. We can eliminate choice 3 by showing that $f_A(1100) \neq Y$. $f_A(1100) \neq Y$ because otherwise 0100 is constructible from the connected set $\{0000, 1000, 1100\}$ and $f_A(0100) = B$ is not in $f_A(\{0000, 1000, 1100\}) = \{R, G, Y\}$, contradicting Lemma 5.2.

We now show how to eliminate choices 4, 5, and 6. We consider each of the three choices in turn.

Case 4. Figure 5(a) shows the current assignment of values to configurations and the possibilities for some currently unassigned configurations. $f_A(0110) \neq G$ because otherwise 0110 is constructible from the connected set $\{0000, 0010, 0100\}$ and $f_A(0110)$ is not in $f_A(\{0000, 0010, 0100\}) = \{R, B, Y\}$, contradicting Lemma 5.2.

Thus, we only have one choice to consider: $f_A(0101) = G$ and $f_A(0110) = Y$. Figure 5(b) shows the current assignment of values to configurations and the possibilities for some currently unassigned configurations. f_A cannot map 0011 to both G and B. This case leads to a dead end.

Case 5. Figure 6(a) shows the current assignment of values to configurations and the possibilities for some currently unassigned configurations. As in Choice 4, $f_A(0110) \neq G$ and thus $f_A(0101) = G$ and $f_A(0110) = Y$. Figure 6(b) shows the current assignment of values to configurations and the possibilities for some currently unassigned configurations. $f_A(1010) \neq B$ because otherwise 1000 is constructible

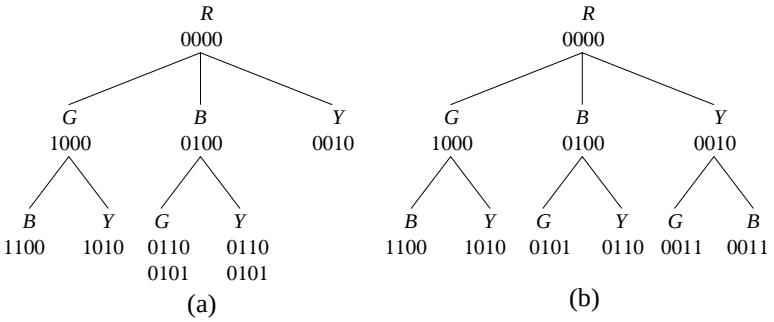

Figure 5: Choices in Case 4 of the Proof of Lemma 5.4

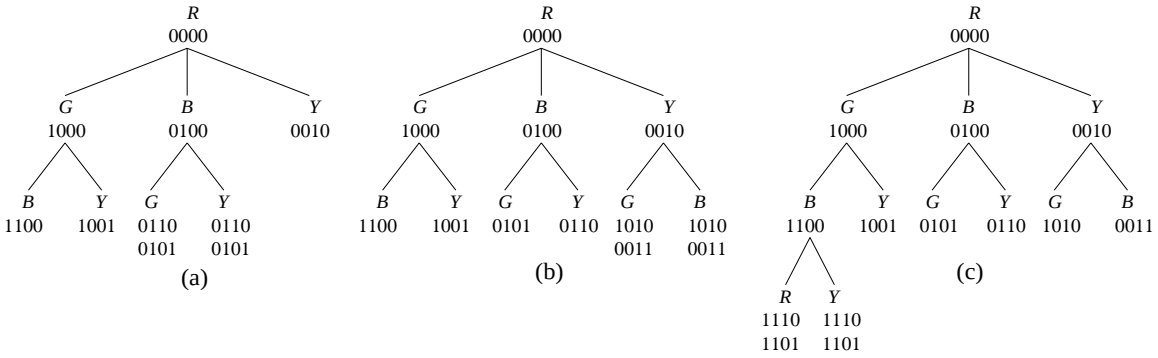

Figure 6: Choices in Case 5 of the Proof of Lemma 5.4

from the connected set $\{0000, 0010, 1010\}$ and $f_A(1000) = G$ is not in $f_A(\{0000, 0010, 1010\}) = \{R, B, Y\}$, contradicting Lemma 5.2.

Thus, we only have one choice to consider: $f_A(1010) = G$ and $f_A(0011) = B$. Figure 6(c) shows the current assignment of values to configurations and the possibilities for some currently unassigned configurations. $f_A(1101) \neq R$ because otherwise 1001 is constructible from the connected set $\{0000, 1000, 1100, 1101\}$ and $f_A(1001) = Y$ is not in $f_A(\{0000, 1000, 1100, 1101\}) = \{R, G, B\}$, contradicting Lemma 5.2. $f_A(1110) \neq R$ because otherwise 0110 is constructible from the connected set $\{0000, 1000, 1100, 1110\}$ and $f(0110) = Y$ is not in $f_A(\{0000, 1000, 1100, 1110\}) = \{R, G, B\}$, contradicting Lemma 5.2. This case leads to a dead end.

Case 6. Figure 7 shows the current assignment of values to configurations and the possibilities for some currently unassigned configurations.

$f_A(0110) \neq G$ because otherwise 1010 is constructible from the connected set $\{0000, 1000, 0100, 0110\}$ and $f_A(1010) = Y$ is not in $f_A(\{0000, 1000, 0100, 0110\}) = \{R, G, B\}$, contradicting Lemma 5.2. $f_A(1100) \neq Y$ because otherwise 1000 is constructible from the connected set $\{0000, 0100, 1100\}$ and $f_A(1000) = G$ is not in $f_A(\{0000, 0100, 1100\}) = \{R, B, Y\}$, contradicting Lemma 5.2. Thus, we have three choices to consider:

6.1. $f_A(1100) = G$ and $f_A(0110) = Y$.

6.2. $f_A(1100) = G$ and $f_A(0101) = Y$.

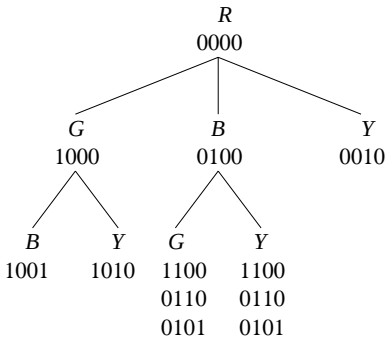

Figure 7: Case 6 and Second Set of Choices in the Proof of Lemma 5.4

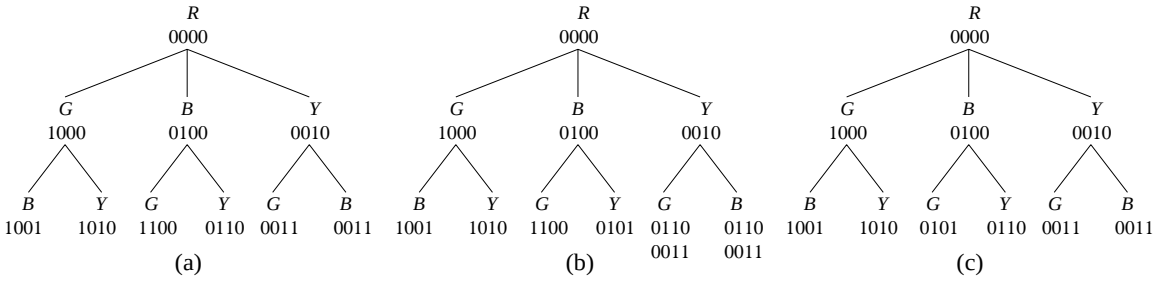

Figure 8: Subcases of Case 6 in the Proof of Lemma 5.4: (a) Case 6.1; (b) Case 6.2; (c) Case 6.3.

6.3. $f_A(0101) = G$ and $f_A(0110) = Y$.

We consider each of the three choices in turn.

Case 6.1. Figure 8(a) shows the current assignment of values to configurations and the possibilities for some currently unassigned configurations. f_A cannot map 0011 to both G and B. This choice leads to a dead end.

Case 6.2. Figure 8(b) shows the current assignment of values to configurations and the possibilities for some currently unassigned configurations. $f_A(0110) \neq G$ because otherwise 0100 is constructible from the connected set $\{0000, 0010, 0110\}$ and $f_A(0100) = B$ is not in $f_A(\{0000, 0010, 0110\}) = \{R, G, Y\}$, contradicting Lemma 5.2. $f_A(0011) \neq G$ because otherwise 1001 is constructible from the connected set $\{0000, 1000, 0010, 0011\}$ and $f_A(1001) = B$ is not in $f_A(\{0000, 1000, 0010, 0011\}) = \{R, G, Y\}$, contradicting Lemma 5.2. This case leads to a dead end.

Case 6.3. Figure 8(c) shows the current assignment of values to configurations and the possibilities for some currently unassigned configurations. f_A cannot map 0011 to both G and B. We have nowhere else to backtrack.

Thus, $R(4) > 4$. ■

Lemma 5.5 *Consider any $A \in SYM(k)$ with $R_A = m$. Suppose there exists a reachable configuration C*

and a value $w \neq f_A(C)$ such that C has p neighbors D with $f_A(D) = w$. Then there exists an algorithm $A' \in SYM(k-1)$ with $R_{A'} \leq m - p$.

Proof We show how to construct A' given A. A' will implement a logical register with value set $V - \{w\}$, where V is the value set of the logical register implemented by A, and initial value $v_0 \in V - \{w\}$.

Let $\{C_1, C_2, \ldots, C_p\}$ be the p neighbors of C such that $f_A(C_i) = w$, for each $i \in \{1, \ldots, p\}$. For each i, let b_i be the bit in which C and C_i differ. Consider the set S of all configurations reachable from C by a path of algorithm edges in which no configuration X with $f_A(X) = w$ appears in the path. Let Z be the subgraph of H_A in which the node set is S and the edge set is the set of all edges in $S \times S$ that are algorithm edges in H_A. No edge in Z is labeled with any bit in $\{b_1, b_2, \ldots, b_p\}$ because otherwise some C_i is constructible from S, which is connected, and $f_A(C_i) = w$ is not in $f_A(S)$, contradicting Lemma 5.2.

Algorithm A' will use $m - p$ binary regular registers: the same m registers used in algorithm A except for the p registers $\{b_1, b_2, \ldots, b_p\}$. We now describe the reachable configurations of A'. Assume without loss of generality that b_1 through b_p are the last p bits and they are all 0 in C. Thus, b_1 through b_p are all 0 in every configuration in S. Given $D \in S$, define $\pi(D)$ to be the prefix of D consisting of all but the last p bits. These will be the reachable configurations of A'. If $f_A(C) = v_0$, let $D_0 = C$. Otherwise, let D_0 be the neighbor of C in Z such that $f_A(D_0) = v_0$. Clearly D_0 exists. We define the initial configuration of A' to be $\pi(D_0)$.

We now describe the reader's protocol in algorithm A'. The reader's protocol in algorithm A' is the same as the reader's protocol in algorithm A, except that the reader in A' has local bits $c_1, \ldots, c_p$ corresponding to shared bits $b_1, \ldots, b_p$ in A. The value of bit c_i is 0 for each $i \in \{1, \ldots, p\}$ at all times. Whenever reader j in A reads shared bit b_i, the reader in A' reads its local bit c_i.

We now describe the writer's protocol in algorithm A'. If the current configuration of the physical registers (well-defined because readers do not write) is $\pi(E)$ for some $E \in S$ and if WRITE(x), for x not the current value of the logical register, is the next operation, then the writer changes bit b, where b labels the algorithm edge (E, D) in Z and $f_A(D) = x$. Since $x \neq w$, none of the bits in $\{b_1, \ldots, b_p\}$ are changed, so we end up in configuration $\pi(D)$. An easy induction shows that in every state of every execution of A' the physical registers always form a configuration $\overline{E}$ such that $\overline{E} = \pi(E)$ for some $E \in S$.

Now we must show that algorithm A' implements a $(k-1)$-ary regular register. Algorithm A' clearly holds $(k-1)$ values and satisfies the wait-free property. We now show that the regular property holds. Consider any execution e' of algorithm A'. We build a corresponding execution e of algorithm A as follows. We construct a sequence of actions of A by starting with a sequence of logical WRITEs (called the initial WRITEs) to ensure that the configuration of the physical registers is D_0. We then consider each action in the execution of A' in turn. Let j be the process performing the action. First, the action is placed in the sequence. Then, for each read of a local bit c_i that occurs in e' during the execution of this action, we place the actions $\text{read}_{b_i}(j)$ and $\text{return}_{b_i}(j, 0)$. Note that there are no writes into a local bit c_i in any execution of A'.

By induction, there exists an execution e of A with the sequence of actions just constructed. By the assumption about A, e satisfies the regular property. Suppose a READ by reader j in execution e' of algorithm A' RETURNs value v. Then the corresponding READ in the constructed execution e of algorithm A also RETURNs value v. We must prove that v is a proper value to RETURN in e'. In e, v is the value of an overlapping WRITE, the value of the last preceding WRITE, or the initial value of A. We consider each possibility in turn. If in e, v is the value of an overlapping WRITE, then WRITE(v) overlaps the original READ in e'. Thus v is a proper value to RETURN in e'. If in e, v is the value of the last preceding WRITE W, then there are two possibilities to consider, depending on W. If W is not an initial WRITE, it must have a corresponding WRITE(v) in e', and v is a proper value to RETURN in e'. If W is an initial WRITE, it must in fact be the last initial WRITE, writing v_0, the initial value for A'. In this case there is no WRITE that precedes the READ in e', and the READ RETURNs v_0, which is the proper value. If in e, v is the initial value of A and no WRITE precedes the READ, then the initial value of A is also v_0 (there are no initial WRITEs) and the READ in e' has no preceding WRITE. Thus v is a proper value to RETURN in e'. Therefore algorithm A' satisfies the regular property.

A' satisfies the symmetric property because A satisfies the symmetric property, and $R_{A'} \leq m - p$. ■

Lemma 5.6 $R(k-1) \leq R(k) - 1$.

Proof Let $R(k) = m$ and choose any $A \in \text{SYM}(k)$ with $R_A = R(k)$. Let C be a reachable configuration of A. Since A is a one-write algorithm, C has a neighbor D such that $f_A(D) \neq f_A(C)$. By Lemma 5.5 with $p = 1$, there exists an $A' \in \text{SYM}(k-1)$ with $R_{A'} \leq m - 1$. Thus $R(k-1) \leq m - 1$, implying that $R(k-1) \leq R(k) - 1$. ■

Lemma 5.7 *If k is not a power of 2, then $R(k-1) \leq R(k) - 2$.*

Proof Let $R(k) = m$, where k is not a power of 2, and choose any $A \in \text{SYM}(k)$ with $R_A = R(k)$. We need to show that there exists a reachable configuration C, some $w \neq f_A(C)$, and at least two neighbors D_1 and D_2 of C such that $f_A(D_1) = f_A(D_2) = w$. The result would then follow from Lemma 5.5, substituting 2 for p. The rest of this proof is devoted to showing that such a configuration exists. Assume for contradiction that for every reachable configuration C and every $w \neq f_A(C)$, C has at most one neighbor D with $f_A(D) = w$. For sake of clarity, we call this the one-neighbor assumption.

Claim 5.8 *For any reachable C, f_A maps all unreachable neighbors of C to $f_A(C)$.*

> **Proof** Suppose in contradiction that C has one unreachable neighbor E such that $f_A(E) \neq f_A(C)$. C already has a reachable neighbor D with $f_A(D) = f_A(E)$ because A is a one-write algorithm. This means that C has at least two neighbors mapped by f_A to $f_A(E)$, a contradiction to the one-neighbor assumption.
>
> **End of Claim**

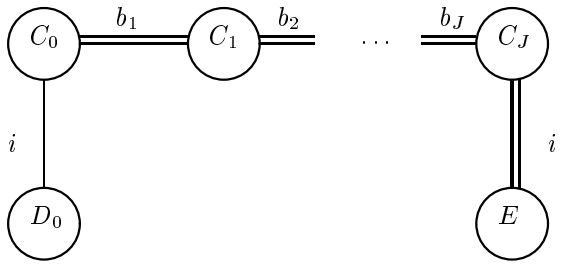

Figure 9: Relationships Among the Configurations in the Chain from C_0 to E

Claim 5.9 *All configurations are reachable.*

Proof Suppose in contradiction that there exists an unreachable configuration. Then there exists a reachable configuration C_0 that has an unreachable neighbor D_0. $f_A(D_0) = f_A(C_0)$ by Claim 5.8. Suppose D_0 and C_0 differ only in bit i. Since we are assuming that the minimum number of binary regular registers is used, there exists some reachable configuration E such that E and C_0 differ in bit i and bit i labels the last edge in some path of algorithm edges in H_A connecting C_0 and E. The length of the path from C_0 to E must be at least 2. Let the bits that changed in the path from C_0 to E be $b_1, b_2, \ldots, b_J, i$, in that order and let the sequence of configurations in the path be $C_0, C_1, C_2, \ldots, C_J, E$. Then C_J and E differ only in bit i. Figure 9 shows the relationships among these configurations where double lines denote algorithm edges and single lines denote non-algorithm edges. For all j, $1 \leq j \leq J$, let D_j be the neighbor of C_j that differs from C_j in bit i. Notice that D_0 is unreachable, and $D_J = E$, which is reachable. Since $D_0, D_1, \ldots, D_J = E$ is the sequence of configurations in some path (the bits that change in this path are $b_1, b_2, \ldots, b_J$, in that order), there exists a j such that D_{j-1} is unreachable and D_j is reachable. Figure 10 shows the relationships among C_{j-1}, C_j, D_{j-1}, and D_j. Dashed lines denote edges that are not known to be either algorithm or non-algorithm edges. Let $f_A(C_{j-1}) = v_1$. $f_A(C_j) \neq v_1$ because (C_{j-1}, C_j) is an algorithm edge. Since D_{j-1} is an unreachable neighbor of reachable C_{j-1}, $f_A(D_{j-1}) = f_A(C_{j-1}) = v_1$ by Claim 5.8. Similarly, since D_{j-1} is an unreachable neighbor of reachable D_j, $f_A(D_j) = f_A(D_{j-1}) = v_1$ by Claim 5.8. Thus C_j has two neighbors C_{j-1} and D_j mapped by f_A to v_1, a contradiction to the one-neighbor assumption.

End of Claim

Choose some $v \in V$. Let b be the number of configurations C with $f_A(C) = v$. Let B be the set of edges (C, D) such that either $f_A(C) = v$ and $f_A(D) \neq v$ or $f_A(C) \neq v$ and $f_A(D) = v$.

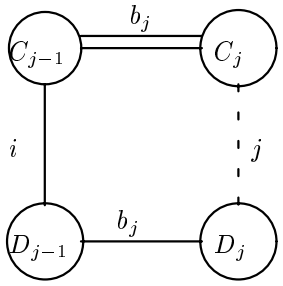

Figure 10: Relationships Among C_{j-1}, C_j, D_{j-1}, and D_j

For any configuration C, we know that C is reachable by Claim 5.9. Now, by the one-neighbor assumption, for every $w \neq f_A(C)$, C has at most one neighbor D with $f_A(D) = w$. Since A is a one-write algorithm, it follows that C has exactly one neighbor D with $f_A(D) = w$ for every $w \neq f_A(C)$. Therefore, for each configuration C such that $f_A(C) = v$, C has exactly $k-1$ neighbors D with $f_A(D) \neq v$. This implies that $|B| = b(k-1)$. Also, for each configuration C such that $f_A(C) \neq v$, C has exactly one neighbor D with $f_A(D) = v$. This implies that $|B| = 2^m - b$. Then $2^m - b = b(k-1)$, which implies that $2^m = kb$, which means that k is a power of 2. This contradicts the fact that k is not a power of 2. ■

Theorem 5.10 *For all $k \geq 4$, $R(k) > 2k - 2 - \lfloor \log k \rfloor$.*

Proof We proceed by induction on k.

Basis: $k = 4$. $2k - 2 - \lfloor \log k \rfloor = 4$. By Lemma 5.4, $R(4) > 4$.

Inductive step: $k > 4$. Suppose the lemma is true for $k-1$. Now we show that it is true for k. There are two possibilities for k. Either k is a power of 2, or k is not a power of 2.

Case 1: k is a power of 2.

$$\begin{aligned} R(k) &\geq R(k-1) + 1 && \text{by Lemma 5.6} \\ &> 2(k-1) - 2 - \lfloor \log(k-1) \rfloor + 1 && \text{by the inductive hypothesis} \\ &= 2k - 2 - 2 - (\lfloor \log k \rfloor - 1) + 1 && \text{because } k \text{ is a power of 2} \\ &= 2k - 2 - \lfloor \log k \rfloor. \end{aligned}$$

Case 2: k is not a power of 2.

$$\begin{aligned} R(k) &\geq R(k-1) + 2 && \text{by Lemma 5.7} \\ &> 2(k-1) - 2 - \lfloor \log(k-1) \rfloor + 2 && \text{by the inductive hypothesis} \\ &= 2(k-1) - 2 - \lfloor \log k \rfloor + 2 && \text{because } k \text{ is not a power of 2} \\ &= 2k - 2 - \lfloor \log k \rfloor. \end{aligned}$$

■

5.4 Toggle Property

A normal form one-write algorithm has the **toggle property** if for each pair of distinct $v, w \in V$, there exists a bit l such that whenever the value of the logical register is changed from v to w or from w to v, bit l is written. A one-write algorithm satisfying the toggle property clearly satisfies the symmetric property and thus the undirected version of Lemma 5.2 holds.

Our algorithm satisfies the toggle property. We will show that our algorithm is optimal in the class of algorithms satisfying this property with respect to the number of physical registers.

Every algorithm A with the toggle property can be represented by the complete graph on k nodes, in which each node is labeled with a distinct element from V and the edge between v and w is labeled with some $l \in \{1, \ldots, m\}$ (when the value of the logical register is changed from v to w or vice versa, bit l is changed), where m is the number of binary registers used by A. Call this graph G_A.

When $k = 3$, our algorithm is optimal in the number of binary regular registers used because $C(k, 2)$ matches the lower bound of k from [2]. Theorem 5.11 below shows that $C(k, 2)$ binary regular registers are necessary for any $k \geq 4$.

Theorem 5.11 *For all normal form one-write algorithms A for implementing a k-ary ($k \geq 4$) regular register from binary regular registers, if A has the toggle property, then the number of binary regular registers used by A is at least $C(k, 2)$.*

Proof Suppose that A is a one-write algorithm for implementing a k-ary regular register from binary regular registers, where A has the toggle property and the number of registers used by A is less than $C(k, 2)$. Then there is some register i such that i is the label of at least two edges in G_A, say (v_1, v_2) and (v_3, v_4). Suppose the edges have a common endpoint. Without loss of generality, assume $v_1 = v_3$. Then $v_2 \neq v_4$ because otherwise the edges would be the same. If the current value of the logical register is v_1 and bit i is changed, the new value of the logical register is both v_2 and v_4, which is ambiguous. Thus the edges are disjoint; v_1, v_2, v_3, and v_4 are distinct.

Let j, where $j \neq i$, label the edge (v_1, v_3) of G_A. Let C_1 be any configuration such that $f_A(C_1) = v_1$. Let C_2 be the configuration that differs from C_1 only in bit i. Let C_3 be the configuration that differs from C_1 only in bit j. Let C_4 be the configuration that differs from C_1 only in bits i and j. By the definition of G_A, C_2, C_3, and C_4 are reachable configurations, and $f_A(C_2) = v_2$, $f_A(C_3) = v_3$, and $f_A(C_4) = v_4$. Figure 11 shows the relationships among C_1, C_2, C_3, and C_4. C_2 is constructible from the connected set $\{C_1, C_3, C_4\}$. But $f_A(C_2) = v_2$ is not in $f_A(\{C_1, C_3, C_4\}) = \{v_1, v_3, v_4\}$, contradicting Lemma 5.2. ■

5.5 Atomicity

In this subsection we establish lower bounds on the number of registers required by two classes of atomic one-write algorithms. Since atomicity is a stronger property than regularity, we may not be able to transform

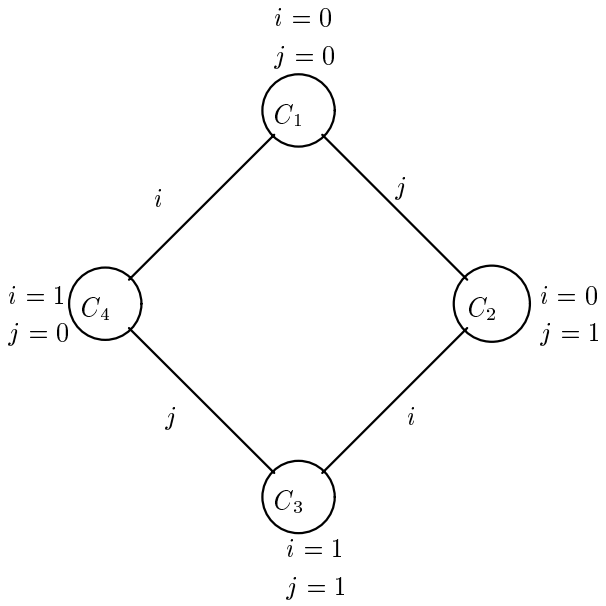

Figure 11: Relationships Among the Four Configurations in the Proof of Theorem 5.11

an arbitrary atomic one-write algorithm to a normal form one-write algorithm. It may help for readers to communicate with each other and the writer by writing to binary registers. Similarly, we may not be able to transform an arbitrary atomic one-write algorithm to one in which the reader does not read registers more than once. Since atomicity implies regularity, Theorems 5.10 and 5.11 are true for normal form one-write atomic algorithms in which readers read registers only once. The proofs for the atomic case are identical to the proofs of Theorems 5.10 and 5.11.

6 Conclusion

We have proven the existence of a one-write algorithm for implementing a k-ary regular register from binary regular registers. The same algorithm implements a k-ary atomic register from binary atomic registers. The algorithm we have developed uses $k(k-1)/2$ binary registers. It is optimal in the number of binary registers used with respect to all one-write algorithms satisfying the toggle property. We have also improved the lower bound on the number of binary registers required for all one-write algorithms satisfying the symmetric property from k to $2k-1-\lfloor \log k \rfloor$. Our lower bound proofs are modular, and they use our general technique for "fooling the reader." We also made simplifying assumptions about the readers' programs and showed the simplifications did not lead to any loss of generality.

An interesting open question is to determine tight bounds on the number of physical registers needed for symmetric algorithms and more general types of algorithms. Lemma 5.5, which is our general algorithm

transformation technique, may help in obtaining tighter bounds. For example, if one can establish that $p = \Theta(\log k)$, then one can obtain a lower bound of $\Omega(k \log k)$ registers. Another interesting open question is to improve the known lower bound on the number of registers a reader must read.

7 Acknowledgments

We thank Michael Fischer for his helpful comments that greatly improved the presentation, especially in Section 4.2.

References

[1] Soma Chaudhuri, Martha J. Kosa, and Jennifer L. Welch. Upper and Lower Bounds for One-Write Multivalued Regular Registers. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, December 1991. Also available as TR91-026 from the University of North Carolina at Chapel Hill.

[2] Soma Chaudhuri and Jennifer L. Welch. Bounds on the Costs of Register Implementations. *SIAM Journal on Computing*, 23 (2), April, 1994.

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

[4] Prasad Jayanti, Adarshpal Sethi, and Errol L. Lloyd. Minimal Shared Information for Concurrent Reading and Writing. In *Proceedings of the Fifth International Workshop on Distributed Algorithms*, October 1991.

[5] Lefteris Kirousis and Evangelos Kranakis. A Survey of Concurrent Readers and Writers. *CWI Quarterly*, 2:307–330, 1989.

[6] Leslie Lamport. On Interprocess Communication. *Distributed Computing*, 1(1):86–101, 1986.

[7] Nancy A. Lynch and Mark R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. In *Proceedings of the Sixth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 137–151, August 1987.