SIAM J. COMPUT.
Vol. 23, No. 2, pp. 335–354, April 1994

BOUNDS ON THE COSTS OF MULTIVALUED REGISTER IMPLEMENTATIONS*

SOMA CHAUDHURI† AND JENNIFER L. WELCH‡

Abstract. A fundamental aspect of any concurrent system is how processes communicate with each other. Ultimately, all communication involves concurrent reads and writes of shared memory cells, or *registers*. The stronger the guarantees provided by a register, the more useful it is to the user, but the harder it may be to implement in practice. This paper considers the problem of implementing a k-ary regular (respectively, safe) register out of binary regular (respectively, safe) registers, assuming a single writer. While algorithms have been developed previously for these problems, no nontrivial lower bounds were known. The cost measures considered here are the number of physical registers and the number of reads and writes on the physical registers required to implement the logical register. Tight bounds are obtained on the cost measures in many cases, and interesting trade-offs between the cost measures are identified. The lower bounds are shown using information-theoretic techniques. Two new algorithms are presented that improve on the costs of previously known algorithms: the *hypercube* algorithm implements a k-ary safe register out of binary safe registers, requiring only one physical write per logical write; and the *tree* algorithm implements a k-ary regular register out of binary regular registers, requiring only $\lceil \log k \rceil$ physical operations per logical operation. Both algorithms use novel combinatorial techniques.

Key words. registers, concurrent distributed system, concurrent computation, shared memory registers, time and space complexity

AMS subject classifications. 68Q22, 68Q25

1. Introduction.

A fundamental aspect of any concurrent system is how processes communicate with each other. Ultimately, all communication involves concurrent accesses to shared memory cells, or registers. The stronger the guarantees provided by the shared memory, the more useful it is to the user, but the harder it may be to implement in practice. Thus it is of interest to determine which types of registers can implement which other types. Many such implementations are known, e.g., [1], [2], [6], [7], [10], [11], [12], [13], [15], [16], [17], among many others.

The contribution of this paper is to study the *costs* of implementing one type of register (the *logical* register) out of registers of another type (the *physical* registers). Cost measures considered are the number of physical registers and the number of operations on the physical registers used to perform the operations of the implemented register. Bounds on the number of physical operations can be used to obtain time bounds for the logical operations in terms of the time taken by the physical operations.

A *register* is a shared variable or memory cell that supports concurrent reading and writing by a collection of processing entities. The operations of reading and writing are not instantaneous; instead, they have duration in time, from a starting point to an ending point. Although each entity accessing a register is assumed to issue operations sequentially, operations on behalf of different entities can overlap in time.

A variety of types of registers can be defined, differing in several dimensions, including the number of concurrent readers supported, the number of concurrent writers supported, the number of values the register can take on, and the strength of the consistency guarantees provided in the presence of concurrent operations. Throughout this paper we assume there

*Received by the editors November 19, 1990; accepted for publication (in revised form) November 30, 1992. This work was done while the authors were at the Department of Computer Science, University of North Carolina at Chapel Hill, and supported in part by National Science Foundation grant CCR-9010730, an IBM Faculty Development Award, and National Science Foundation Presidential Young Investigator Award CCR-9158478.

†Department of Computer Science, Iowa State University, Ames, Iowa 50011 (chaudhur@cs.iastate.edu).

‡Department of Computer Science, Texas A&M University, College Station, Texas 77843 (welch@cs.tamu.edu).

is only one writer, leaving three parameters of interest: the number of readers, the number of values, and the consistency guarantees. We distinguish between 1-reader registers and n-reader registers, for $n > 1$, and between binary registers and k-ary registers, for $k > 2$. (A k-ary register can take on k different values.)

Lamport [6] defines three consistency guarantees of increasing strength, namely, safe, regular, and atomic. Roughly speaking, a read of a *safe* register always returns the most recent value written to the register, unless the read overlaps with a write, in which case any legal value of the register can be returned. A read of a *regular* register always returns the most recent value written, unless the read overlaps one or more writes, in which case it returns either the old value or one of the values written by an overlapping write. An *atomic* register provides the illusion, via the values returned by read operations, that each operation happens at a single instant in time within its range, i.e., that the operations are totally ordered. In this paper, we only consider safe and regular registers. In particular, we consider the problem of implementing an n-reader k-ary regular (respectively, safe) register out of n-reader binary regular (respectively, safe) registers.

We study the costs incurred by these implementations. Let M, R, and W be the minima, over all implementations between two particular types of registers, of the number of physical registers, the maximum number of physical operations in a logical read, and the maximum number of physical operations in a logical write, respectively. Our algorithms will involve no physical reads in a logical write and no physical writes in a logical read. Our lower bound results give bounds on the number of physical reads per logical read, and the number of physical writes per logical write. These are stronger results than just giving bounds on the number of physical operations per logical action.

Our results are summarized in Tables 1 and 2. Table 1 gives the bounds when all algorithms are considered. Table 2 gives the bounds when certain classes of algorithms are considered, as specified by the column labeled S—namely, 1-write algorithms, c-write algorithms, and $\lceil \log k \rceil$-register algorithms. (All logarithms are base 2.)

For implementing a k-ary safe register out of binary safe registers, we show tight bounds of $R = \lceil \log k \rceil$, $W = 1$, and $M = \lceil \log k \rceil$. The upper bound of 1 on W is obtained from a new algorithm, which we call the *hypercube* algorithm. The best previous upper bound on W was $\lceil \log k \rceil$ [6]. These three optimal bounds are not obtained simultaneously in a single algorithm, and in fact, we show some nontrivial trade-offs between the three cost measures.

For implementing a k-ary regular register out of binary regular registers, we show the tight bound that $R = \lceil \log k \rceil$, and the bounds $1 \leq W \leq \lceil \log k \rceil$, and $\max\{\lceil \log k \rceil + 1, 2(\log k) - \log\log k - 2\} \leq M \leq \min\{k - 1, n(3 \log k + 68)\}$, where n is the number of readers of the logical register. The upper bounds on R and W are simultaneously achieved by a new algorithm, which we call the *tree* algorithm. We also present some lower bounds on R and M that follow if we restrict attention to implementations that use only a small constant number of physical writes per logical write.

The lower bounds in Table 1 for safe registers and those on R and W for regular registers are obvious from information-theoretic considerations. All of the remaining lower bounds are new. Little previous work has been done concerning lower bounds or trade-offs for register implementations. One such previous result is in [6], where it is shown that in any implementation of an atomic register using regular registers, a read of the logical register must involve a write to a physical register. Tromp [13] uses this result to show that three binary safe registers are necessary to construct a binary atomic register.

In §2 we present our model and some results for all implementations. Section 3 considers safe registers and §4 considers regular registers. We conclude in §5 with some open questions.

2. Preliminaries.

In this section, we give formal definitions for the types of registers that we will study (n-reader, k-ary, safe, and regular), describe the rules we impose on implementing

TABLE 1
Independent bounds for binary to k-ary algorithms.

	Safe		*Regular*	
	lower	*upper*	*lower*	*upper*
R	$\lceil \log k \rceil$	$\lceil \log k \rceil$	$\lceil \log k \rceil$	$\lceil \log k \rceil$
W	1	1	1	$\lceil \log k \rceil$
M	$\lceil \log k \rceil$	$\lceil \log k \rceil$	$\max\{\lceil \log k \rceil + 1, \lceil 2 \log k - \log \log k \rceil - 2\}$	$\min\{k - 1, n(3 \log k + 68)\}$

TABLE 2
Trade-off results for binary to k-ary algorithms.

S		*Safe*		*Regular*	
		lower	*upper*	*lower*	*upper*
$\{A \mid W_A = 1\}$	R_S	$k - 1$	$2^{\lceil \log k \rceil} - 1$	$k - 1$	∞
	M_S	$k - 1$ or k[1]	$2^{\lceil \log k \rceil} - 1$	k	∞
$\{A \mid W_A = c\}$	R_S	$(c!k/2)^{1/c}$	$c - 2 + \lceil k/2^{c-2} \rceil$	$(c!k/2)^{1/c}$	∞
	M_S	$(c!k/2)^{1/c}$	$c - 2 + \lceil k/2^{c-2} \rceil$	$(c!k/2)^{1/c}$	∞
$\{A \mid M_A = \lceil \log k \rceil\}$	W_S	$\lceil \log k \rceil$	$\lceil \log k \rceil$	∞	∞

one type of register with another, and define the cost measures we will use. Then we present some definitions and lemmas that are true for implementations between any types of registers.

2.1. Model. We use a simplified form of the I/O automaton model [9] 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; 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.

[1] $k - 1$ if k is a power of 2; k otherwise.

The **physical actions** are $\text{read}_j(i)$, $\text{return}_j(i, v)$, $\text{write}_j(v)$, 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. For a fixed j, the parameter i for reads and returns ranges over some subset of $\{0, \dots, n\}$ of size at most n; this subset indicates which of the read and write processes read X_j. (The value 0 indicates WP.) 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.

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, $1 \leq j \leq m$, $\text{read}_j(i)$ and $\text{return}_j(i, *)$ 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.

Each physical register X_j satisfies this liveness property: Immediately after an input action occurs, the matching output is enabled.

A **safe** physical register satisfies the **Safe Property**: For every physical read operation that does not overlap a physical write operation, the value returned is the value written by the most recent physical write operation. If there is no preceding write operation, then it returns the initial value.

A **regular** physical register satisfies the **Regular Property**: Every 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).

The read and write processes must work together to implement a logical register. The liveness property for a logical register differs from that for a physical register, as discussed below. A safe (respectively, regular) logical register satisfies the safe (respectively, regular) property, as defined for physical registers, replacing "physical" with "logical."

The liveness property for a logical register is that the implementation must be **wait-free**, meaning that in every finite execution, if a logical operation by RP_i (respectively, WP) is pending, then there is a finite sequence of actions involving only RP_i (respectively, WP) that finishes the operation. Our algorithms actually provide a bounded number of actions, while our lower bounds hold for algorithms satisfying the weaker definition.

A natural question that may arise is why the liveness property is different for physical and logical registers. The wait-free definition for the logical register implies that every logical operation must complete using only physical operations initiated by that logical operation. In the case of the physical register, where we don't model the "internal" actions, this wait-free property reduces to the physical liveness property given.

We now define the cost measures.

Consider two register types, physical and logical, and let A be an algorithm for a physical-to-logical register implementation. Let M_A be the number of physical registers used in A, let R_A be the maximum number of physical operations performed during any logical READ in any execution of A, and let W_A be the maximum number of physical operations performed during any logical WRITE in any execution of A. Given a set S of physical-to-logical register implementations, let M_S be the minimum of M_A over all $A \in S$, R_S be the minimum of R_A over all $A \in S$, and W_S be the minimum of W_A over all $A \in S$. (If $S = \emptyset$, then M_S, R_S, and W_S are infinity.) Finally, let $M = M_S$, $R = R_S$, and $W = W_S$, where S is the set of all physical-to-logical register implementations (for these two types). (The physical and logical register types are implicit parameters to M, R, and W.)

In the rest of this paper, we derive upper and lower bounds on M, R, and W, and trade-offs between them, for different physical and logical register types.

These bounds on R and W can be converted into time bounds for performing logical operations as follows. Suppose we know bounds R_l, R_u, W_l, and W_u such that $R_l \leq R \leq R_u$ and $W_l \leq W \leq W_u$. Let r be an upper bound on the time to read a physical register and let w be an upper bound on the time to write a physical register. Let s be an upper bound on the time for a read or write process to perform an action once it becomes enabled. Our upper bounds on R and W come from algorithms, all of which have the property that no logical READ involves a physical write and no logical WRITE involves a physical read. Since we assume that all physical operations are enclosed within logical operations and that only one physical operation can be pending at a time, we deduce that an upper bound on the worst case time to perform a READ of a logical register that is implemented with physical registers is $R_u(r + s) + s$. Similarly, an upper bound on the worst case time to perform a WRITE of a logical register that is implemented with physical registers is $W_u(w + s) + s$. Our lower bounds on R and W do not assume that logical READs do not involve physical writes, or that logical WRITEs do not involve physical reads, and thus they imply analogous lower bounds on the worst case times.

2.2. General results. Given a finite schedule σ of an algorithm A, let the **configuration** of σ be the tuple of sets of "possible values" of the physical registers at the end of the schedule, i.e., if X_i is the ith physical register, then the ith element of the configuration is the set of all values that could be returned by a physical read of that register at that point, according to the safe/regular property. A configuration is **stable** if each element of the tuple is a singleton set. Thus it can be represented as $x_1 \ldots x_m$, where x_i is the value of register X_i for all i. The **initial configuration** is the (stable) configuration of the empty schedule, consisting of the initial value of each physical register.

Let $\mathcal{WO}$ (for "write-only") be the set of all schedules of A in which only WP takes steps and no physical write is pending. Let $\mathcal{S} = \{C : C$ is the configuration of some $\sigma \in \mathcal{WO}\}$. It is easy to see that all configurations in $\mathcal{S}$ are stable.

For each i, define $L_i : \mathcal{S} \to V$ as follows. $L_i(C)$ is the logical value returned by RP$_i$ when RP$_i$ starts in its local initial state, the physical registers have the values specified in C, and no other read or write process takes a step.

What we would like is a function that returns the value of the logical register when the physical registers are in a given configuration. However, an arbitrary algorithm may have different protocols for different read processes (necessitating our use of a subscript on L), and it may use the history of the read process to determine what value is returned. Thus it might be the case that RP$_i$ returns different values at different times in an execution, even given the same configuration. In order to accommodate such algorithms, we define each L_i specifically when RP$_i$ has taken no steps yet.

The next lemma states that L_i is well defined, i.e., that the current configuration (values of the physical registers) and nothing else determines the value of the logical register (as perceived by RP$_i$). This can be shown by a simple induction on the length of the execution.

LEMMA 2.1. *For any algorithm A, the function L_i is well defined for all i.*

Let $\mathcal{WOC}$ (for "write-only, completed") be the set of all schedules of A in which only WP takes steps and no *logical* WRITE is pending. Let $\mathcal{T} = \{C : C$ is the configuration of some $\sigma \in \mathcal{WOC}\}$. It is easy to see that $\mathcal{T} \subseteq \mathcal{S}$. Every configuration in $\mathcal{T}$ is defined to be a **terminal** configuration.

The next lemma states that if no read process has taken any steps and no logical WRITE is in progress, then each L_i is equal to the value of the most recent WRITE to the logical register.

LEMMA 2.2. *For any algorithm A, if σ is in $\mathcal{WOC}$ with configuration C, then, for all i, $L_i(C)$ equals the value of the most recent* WRITE (*the initial value if σ is empty*).

The previous lemma implies that all the L_i's must agree whenever the argument is in the set $\mathcal{T}$. Thus we define $L : \mathcal{T} \to V$ to be $L(C) = L_i(C)$ for any i. It is easy to see that for each $v \in V$, there is a $C \in \mathcal{T}$ such that $L(C) = v$.

In most of our proofs, we only need to consider situations in which no logical WRITE is pending, and thus we can use the notation L. However, in a few places (notably Lemmas 4.4 and 4.5), we must consider what happens in the middle of a logical WRITE, and thus we must use a specific L_i (we choose L_1 for concreteness).

3. k-ary safe register from binary safe registers.

We consider the problem of implementing an n-reader, k-ary, safe register out of n-reader, binary, safe registers, for any $n \geq 1$, where $k > 2$. Subsection 3.1 is devoted to proving tight, independent bounds on R, W, and M. In §3.2, we present an algorithm A such that $W_A = 1$. We also show some nice combinatorial properties related to one-write algorithms. Subsection 3.3 discusses algorithms that allow c physical accesses per logical WRITE. We also give some additional trade-offs between the cost measures.

Let the value set of the logical register be $V = \{0, \ldots, k-1\}$ with initial value $v_0 \in V$.

3.1. Independent bounds.

The following theorem gives matching upper and lower bounds on R, W, and M.

THEOREM 3.1. *The implementation of an n-reader, k-ary, safe register by n-reader, binary, safe registers gives the following independent bounds*: $R = \lceil \log k \rceil$, $W = 1$, *and* $M = \lceil \log k \rceil$.

Proof. The upper bounds on R and M follow from the binary representation algorithm in [6] described below. The upper bound on W follows from our hypercube algorithm presented in §3.2. The lower bounds on W and M are obvious.

We now show the lower bound on R. For each $v \in V$, there is a schedule σ_v of A of the form

$$\text{WRITE}(v)\ \alpha_v\ \text{ACK READ}(1)\ \beta_v\ \text{RETURN}(1, v),$$

where α_v consists solely of actions of WP and contains no ACK, and β_v consists solely of actions of RP_1 and contains no RETURN.

By the definition of read processes, for all distinct v and w, $\beta_v \neq \beta_w$ and the maximal common prefix of β_v and β_w is immediately followed by a return(0) action from some physical register X in β_v and by a return(1) action from X in β_w (or vice versa); that is to say, RP_1 does the same thing in β_v and β_w until it reads a different value. Let γ_v be the sequence of physical values read in β_v, for all v.

Thus, if $v \neq w$, then the sequence γ_v of physical values read in β_v is not equal to the sequence γ_w of physical values read in β_w. There are k distinct sequences of physical values corresponding to the γ_v's, i.e., k binary strings. Thus at least one string, say that corresponding to γ_v, must have length at least $\lceil \log k \rceil$, implying that β_v contains at least $\lceil \log k \rceil$ physical reads. □

The **binary representation algorithm** in [6] implements an n-reader, k-ary, safe register out of $\lceil \log k \rceil$ n-reader, binary, safe registers. The write process writes the binary representation of the logical value into the physical registers. Each read process reads all the physical registers and returns the logical value whose binary representation was read, as long as the value is less than k. Otherwise, it returns any value less than k. This algorithm implies that $R \leq \lceil \log k \rceil$, $W \leq \lceil \log k \rceil$, and $M \leq \lceil \log k \rceil$. By Theorem 3.1, the number of registers and number of physical reads in the binary representation algorithm are both optimal.

The **unary representation algorithm** presented next shows that $W \leq 2$. There are $k-1$ physical registers, $X_1, \ldots, X_{k-1}$. Logical value 0 is represented when all registers are 0.

Logical value $v \neq 0$ is represented when X_v is 1 and the other registers are 0. Each read process reads registers X_1, X_2, and so on, in order, until reading a 1, and RETURNs logical value v, where X_v is the register that returned 1. If no register returns 1, then 0 is RETURNed. To WRITE logical value v, assuming w is the old value of the logical register, the write process writes 0 to X_w if $w \neq 0$, and writes 1 to X_v if $v \neq 0$.

In the next subsection, we will present an algorithm which brings down the number of physical writes per logical WRITE to 1.

3.2. One-write algorithms.

In this subsection, we discuss the class of one-write algorithms. We show that their existence depends on satisfying a combinatorial coloring property of hypercubes.

Figure 1 presents our new **hypercube algorithm**, which shows that $W \leq 1$; it relies on a function f, which will be defined shortly. For now, assume that k is a power of 2. Later we will show how to remove this restriction.

```
Physical Registers: $X_1, \ldots, X_{k-1}$, initially $f(X_1 \ldots X_{k-1}) = v_0$ and $X_j = 1$ for at most one $j$
Read Process RP$_i$, $1 \leq i \leq n$: variables $x_1, \ldots, x_{k-1}$
        READ($i$):
                for $j := 1$ to $k-1$ do $x_j :=$ read $X_j$ endfor
        RETURN($i$, $f(x_1 \ldots x_{k-1})$)
Write process WP: variables $x_1, \ldots, x_{k-1}$, initially $x_j$ equals the initial value of $X_j$ for all $j$
        WRITE($v$):
                if $v \neq f(x_1 \ldots x_{k-1})$ then
                        write $\overline{x_j}$ to $X_j$, where $j$ is such that $f(x_1 \ldots x_{j-1}\overline{x_j}x_{j+1} \ldots x_{k-1}) = v$
                        $x_j := \overline{x_j}$
                endif
        ACK
```

FIG. 1. *Hypercube algorithm.*

We notice an interesting relationship between the correctness of the hypercube algorithm and coloring the nodes of a $(k-1)$-dimensional hypercube with k colors such that each node has a neighbor with each of the $k-1$ colors other than its own. The following definition and lemmas formalize this idea. (Nodes are labeled with $(k-1)$-bit strings, the colors are elements of V, and the function is the coloring.)

A function g is said to have the **rainbow-coloring property** if $g : \{0,1\}^{k-1} \to V$ such that for all $x \in \{0,1\}^{k-1}$, and for all $v \in V$, if $v \neq g(x)$, then there exists $y \in \{0,1\}^{k-1}$ such that $v = g(y)$ and x and y differ in exactly one bit. That is, every node x has a neighboring node with every color other than x's color.

Lemma 3.2 states that if the function f used in the algorithm has the rainbow-coloring property, the hypercube algorithm correctly implements a k-ary safe register using binary safe registers such that each logical WRITE requires one physical write. The rainbow-coloring property ensures that each READ RETURNs the correct value if it does not overlap a WRITE.

LEMMA 3.2. *If function f has the rainbow-coloring property, then the hypercube algorithm is correct.*

We define a function $f : \{0,1\}^{k-1} \to V$ for use in the algorithm. Lemma 3.3 shows that f has the rainbow-coloring property. For positive integer $i < k$, let $bin(i)$ be the binary representation of i in $\log k$ bits (remember that k is a power of 2). For $x \in \{0,1\}^{k-1}$, let x_i be the ith bit of x, i.e., $x = x_1 x_2 \ldots x_{k-1}$. For all $x \in \{0,1\}^{k-1}$, we define $f(x)$ to be the element of V whose binary representation is $\bigoplus_{x_i=1} bin(i)$, where $\oplus$ represents exclusive-or. This expression consists of $\log k$ bits and thus represents a value in the range 0 to $k-1$, i.e., a value in V.

The intuition behind the coloring function f is that we want to go from a $(k-1)$-bit string, the label of a node in the hypercube, to a $(\log k)$-bit string, indicating one of k colors. Given a node with label x, the color assigned is the one whose binary representation is equal to the exclusive-or of the set of $bin(i)$, for all i such that $x_i = 1$. Note that if two nodes x and y differ in the single bit i, then $f(x) \oplus f(y) = bin(i)$. So, given the color of a node x, we can derive the color of any adjacent node y in a consistent manner.

Given this definition of f, the initial values of the physical registers are all 0, except that if $v_0 \neq 0$ then $X_{v_0} = 1$. The computation of j in the writer's code is $bin(j) = bin(v) \oplus bin(f(x_1 \ldots x_{k-1}))$.

LEMMA 3.3. *The function f defined for the hypercube algorithm* (*when k is a power of* 2) *has the rainbow-coloring property.*

Proof. First we must show that for all $x, y \in \{0,1\}^{k-1}$ that differ in exactly one bit, $f(x) \neq f(y)$. Suppose x and y differ in bit i. Then $f(x) \oplus f(y) = bin(i)$. Since $bin(i) \neq 0^{\log k}$, we are done. Second we must show that for all $x, y, z \in \{0,1\}^{k-1}$ such that $y \neq z$ and y and z both differ from x in exactly one bit, $f(y) \neq f(z)$. This can be shown similarly. These two facts together show that f has the rainbow-coloring property. □

Figure 2 illustrates how our algorithm works in the simple case where $k = 4$. Our hypercube is then a three-dimensional cube, whose vertices can be colored with four colors, r, b, g, and y. Note that the coloring satisfies the rainbow-coloring property.

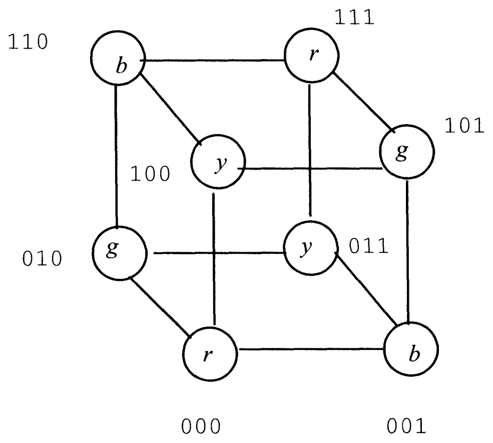

FIG. 2. *An example illustrating the hypercube algorithm.*

Combining Lemmas 3.2 and 3.3 shows that the hypercube algorithm is a one-write algorithm (using $k-1$ registers) if k is a power of 2. To obtain a one-write algorithm for values of k that are not powers of 2, we modify the power-of-2 hypercube algorithm for $m-1$ physical registers, where $m = 2^{\lceil \log k \rceil}$, i.e., m is the smallest power of 2 larger than k. The modification is to change the RETURN statement to be RETURN(min$\{k-1, f(x_1 \ldots x_{m-1})\}$). This implementation of a k-ary register by binary registers will not cause the binary registers to take on all possible 2^{m-1} values, i.e., no stable configuration of the algorithm will be mapped to a value that is out of the range of the logical register. However, a slow read process, which overlaps a number of writes, might (spuriously) observe a configuration corresponding to a value larger than $k-1$, thus necessitating the modification. Thus we have shown the following theorem.

THEOREM 3.4. *The hypercube algorithm correctly implements a k-ary safe register using binary safe registers.*

The following theorem summarizes our results for the class of 1-write algorithms.

THEOREM 3.5. *Let S be the set of algorithms A such that $W_A \leq 1$. Then*

(i) $k-1 \leq R_S \leq 2^{\lceil \log k \rceil} - 1$,

(ii) $M_S = k-1$, *if k is a power of 2, and*

(iii) $k \leq M_S \leq 2^{\lceil \log k \rceil} - 1$, *if k is not a power of 2.*

Proof. All the upper bounds follow from the hypercube algorithm. The rest of the proof concerns the lower bounds.

Choose an algorithm $A \in S$. Let C_{v_0} be the initial configuration. For all $v \neq v_0$, let C_v be the configuration of a schedule in $\mathcal{WOC}$ of the form WRITE(v) α_v ACK, where α_v contains no ACK. Since α_v only contains one physical write, C_{v_0} and C_v differ in a single bit, say that for physical register X_v.

Since there are $k-1$ choices for $v \neq v_0$, there are at least $k-1$ physical registers. Since A was chosen arbitrarily, $M_S \geq k-1$. The improved lower bound of k for M_S when k is not a power of 2 follows from Lemmas 3.6 and 3.7.

To show $R_S \geq k-1$, we assume, for contradiction, that $R_A < k-1$. Consider the schedule READ(1) β RETURN(1, v_0), where β consists solely of actions of RP$_1$ and contains no RETURN. β contains a sequence of less than $k-1$ physical reads. Let X_v (as defined above) be one of the physical registers not read in β; note that $v \neq v_0$. Since C_{v_0} differs from C_v in the value of register X_v and nowhere else, an easy induction on the length of β shows that WRITE(v) α_v ACK READ(1) β RETURN(1, v_0) is a schedule of A, violating the safe property since $v \neq v_0$. We therefore have a contradiction, implying $R_A \geq k-1$. □

We now consider the number of registers when k is not a power of 2. Lemma 3.6, which is the converse of Lemma 3.2, shows that the existence of a function with the rainbow-coloring property is necessary for the existence of a one-write algorithm using $k-1$ registers. Lemma 3.7, which is the converse of Lemma 3.3, shows that when k is not a power of 2, no function with the rainbow-coloring property can exist. Together, these two lemmas imply that if k is not a power of 2, then any one-write algorithm must use more than $k-1$ registers.

LEMMA 3.6. *If there is an algorithm A with $W_A = 1$ and $M_A = k-1$, then there exists a function with the rainbow-coloring property.*

Proof. We show that L has the rainbow-coloring property. Recall that L maps $\mathcal{T}$, the set of terminal configurations, to V. We know $\mathcal{T}$ is not empty. Choose any configuration $C \in \mathcal{T}$. Let v be the color of C under L. Every neighbor of C is also in $\mathcal{T}$ and has a unique color different from C's color, since there are only $k-1$ registers and $k-1$ possibilities for the next value not to be v. To finish the proof, we note that $\mathcal{T} = \{0,1\}^{k-1}$, since every neighbor of a terminal configuration is also a terminal configuration. □

LEMMA 3.7. *If k is not a power of 2, then there is no function with the rainbow-coloring property.*

Proof. Assume in contradiction that there is a function f with the rainbow-coloring property. Choose any color, say blue, and let b be the number of nodes colored blue by f. Let B be the set of edges in the hypercube that have one endpoint colored blue and one endpoint not colored blue. Since each nonblue node is adjacent to exactly one blue node and there are $2^{k-1} - b$ nonblue nodes, $|B|$ must be $2^{k-1} - b$. However, since each blue node is adjacent to $k-1$ nonblue nodes and there are b blue nodes, $|B|$ must be $b(k-1)$. The implication is that $2^{k-1} - b = b(k-1)$, implying $2^{k-1} = kb$. This implies that k divides 2^{k-1}, contradicting the fact that k is not a power of 2. □

In this subsection, we showed that the existence of a one-write implementation of a k-ary safe register was based on solving an underlying combinatorial problem. Specifically, a one-write algorithm using $k-1$ physical registers exists if and only if we can color a $(k-1)$-dimensional hypercube with k colors such that each node has a neighbor with every color other than its own. We can generalize this to any number of physical registers as follows. A one-write algorithm using m registers exists if and only if we can *partially* color an m-dimensional hypercube with k colors, such that each *colored* node has a neighbor with every color other than its own. By a partial coloring, we mean a coloring where not all nodes of the graph need to be colored.

Lemmas 3.3 and 3.7 imply that there is rainbow-coloring of the $(k-1)$-dimensional hypercube if and only if k is a power of 2. Kant and van Leeuwen [5] have independently shown the same result. Their proof uses notions from coding theory and is based on showing a correspondence between 1-error-correcting codes and these colorings. They applied this result to the file distribution problem.

3.3. c-write algorithms and trade-off results. As we showed for 1-write algorithms, the problem of implementing c-write algorithms can also be shown to have a corresponding parallel in a combinatorial problem. Here, we are interested in a partial coloring of the m-dimensional hypercube such that for each colored node, there exists a node of every other color *within a distance of* c from this node.

This combinatorial characterization also helps us obtain lower bounds for M and R. For example, we know that for there to exist a one-write algorithm that uses m physical registers, there must be a configuration C_0 that differs from $k-1$ different configurations C_v in exactly one bit. Since each configuration is represented in m bits, this says that there is a binary string of length m that differs from $k-1$ different strings of the same length in exactly one bit. To satisfy this combinatorial property, we require that $m \geq k-1$. This sequence of reasoning was implicit in the proof of Theorem 3.5.

Along similar lines, for there to exist a c-write algorithm that uses m registers, there must exist a binary string of length m which differs from $k-1$ different strings of the same length in *at most* c bits.

We formalize this property in Lemma 3.8 and Theorems 3.10 and 3.11. These theorems give lower bounds on M and R for c-write algorithms, i.e., algorithms that use a small bounded number of physical writes per logical WRITE.

The next result is Theorem 3.12, which gives trade-offs on W versus R and M. An application of this result is to give upper bounds on M and R for c-write algorithms.

The final result in this subsection, Theorem 3.13, states that if no more than $\lceil \log k \rceil$ registers are used, then some WRITE must write at least $\lceil \log k \rceil$ physical registers.

LEMMA 3.8. *Given any binary string x of length m, if there are at least k distinct strings of length m which differ from x in at most c bits, where $c \leq (\log k)/3$, then $m \geq (c!k/2)^{1/c}$.*

Proof. Let x be a string of length m. The number of distinct strings of length m that differ from x in at most c bits is

$$\binom{m}{0} + \binom{m}{1} + \binom{m}{2} + \cdots + \binom{m}{c}.$$

Since we know that there are at least k such distinct strings, we have $\sum_{i=0}^{c} \binom{m}{i} \geq k$. We obtain the following upper bound on $\binom{m}{i}$, for all i: $\binom{m}{i} = \frac{m(m-1)(m-2)\cdots(m-i+1)}{i!} \leq m^i/i!$. To get an upper bound on the entire summation, we need the following claim, which is taken from [14]. First, we introduce some notation. Let $s_{m,j}$ denote $\sum_{i=0}^{j} \binom{m}{i}$. Let $b_{m,i}$ denote $\binom{m}{i}$.

CLAIM 3.9. *If $1 \leq j \leq m/3$, then $s_{m,j} \leq 2b_{m,j}$.*

Proof. We compute a lower bound for $b_{m,j}/b_{m,j-1} = \frac{m-j+1}{j}$. Note that $\frac{m-j+1}{j}$ is larger than 2 for $j \leq m/3$. Therefore, for $j \leq m/3$, $b_{m,j}/b_{m,j-1} > 2$. The remaining proof is by induction.

Inductive hypothesis: $s_{m,j} \leq 2b_{m,j}$ for $j \leq m/3$.

Basis: For $j = 1$ (assume $m \geq 3$), $s_{m,0} = b_{m,0} = 1$ and $b_{m,1} = m$. Therefore, $s_{m,1} = m + 1$ and $s_{m,1} \leq 2b_{m,1}$.

Inductive step: Let the inductive hypothesis hold for all l such that $l < j \leq m/3$. We show that it holds for j. By the inductive hypothesis, $s_{m,j-1} \leq 2b_{m,j-1}$. Note that $s_{m,j} = s_{m,j-1} + b_{m,j}$. This implies that $s_{m,j} \leq 2b_{m,j-1} + b_{m,j}$. Also, we showed earlier that $2b_{m,j-1} \leq b_{m,j}$. Therefore, $s_{m,j} \leq b_{m,j} + b_{m,j} = 2b_{m,j}$. □

The above claim holds for $j = c$ since we know that $m \geq \log k$ (it takes $\log k$ bits to represent k distinct values), and this implies that $c \leq m/3$. Now, using the above claim and our previous upper bound for $\binom{m}{i}$, we have $\sum_{i=0}^{c} \binom{m}{i} \leq 2\binom{m}{c} \leq 2m^c/c!$. So, $k \leq 2m^c/c!$; by manipulating this inequality, we get $m \geq (c!k/2)^{1/c}$. □

THEOREM 3.10. *For all algorithms A, if $W_A = c$, where $c \leq (\log k)/3$, then $M_A \geq (c!k/2)^{1/c}$.*

Proof. Given an algorithm A such that $W_A = c$, where $c \leq (\log k)/3$, let C_{v_0} be the initial configuration. Then $L(C_{v_0}) = v_0$. For all $v \neq v_0$, the schedule σ_v of the form WRITE(v) α_v ACK yields the terminal configuration C_v. Since each WRITE can initiate at most c physical writes, each C_v differs in at most c bits from C_{v_0}.

Since there are k values v, there must be at least k terminal configurations C_v differing in at most c bits from C_{v_0}. The number of registers used in the algorithm is M_A. Each terminal configuration is therefore a binary string of length M_A. Therefore, there are at least k strings of length M_A which differ in at most c bits from C_{v_0}. Since, $c \leq (\log k)/3$, Lemma 3.8 applies, and we have the result $M_A \geq (c!k/2)^{1/c}$. □

THEOREM 3.11. *For all algorithms A, if $W_A = c$, where $c \leq (\log k)/3$, then $R_A \geq (c!k/2)^{1/c}$.*

Proof. For any algorithm A, where $W_A \leq c$, consider the following schedules, for all v,

$$\text{WRITE}(v)\ \alpha_v\ \text{ACK READ}(1)\ \beta_v\ \text{RETURN}(1, v),$$

where α_v and β_v contain only physical actions. We claim that for some v, β_v initiates at least $(c!k/2)^{1/c}$ physical reads. We prove this by contradiction.

Suppose, for every v, β_v initiates at most p physical reads where $p < (c!k/2)^{1/c}$. Let ρ_v be the sequence of values read, in order, on accessing any given register *for the first time* in β_v. Note that we don't include values obtained from registers which have been read before or been written before in β_v. Clearly, $|\rho_v| \leq p$.

First we assume that the initial configuration is the zero-vector. Therefore, the initial values of all the physical registers are 0. Since α_v contains at most c physical writes, there can be at most c 1's in ρ_v. Clearly, each ρ_v is distinct. Otherwise, if for some v, v' such that $v \neq v'$, $\rho_v = \rho_{v'}$, then a READ in both cases would RETURN the same value, which would be a contradiction. Therefore, $\{\rho_v | v \in V\}$ is a set of k distinct strings of length at most p that differ from the zero-vector in at most c bits. Since $p < (c!k/2)^{1/c}$, this contradicts Lemma 3.8.

In the general case where the initial configuration is *not* the zero-vector, we can no longer claim that ρ_v contains at most c 1's. We therefore define the string δ_v, for every v, as follows. For every bit in ρ_v, if the value is the *same* as the initial value of the register read, place the bit 0 in δ_v. If the value is *different* from the initial value of the register read, place the bit 1 in δ_v. Since α_v contains at most c writes, δ_v can contain at most c 1's. Also, each δ_v is distinct. Now, the same argument as in the previous paragraph holds, with δ_v substituted for ρ_v.

Therefore, for some v, β_v initiates at least $(c!k/2)^{1/c}$ physical reads. This gives our lower bound for R_A. □

Theorem 3.12 presents bounds on the costs of algorithms that are a hybrid of the binary and unary representation algorithms. Using this theorem, we can derive upper bounds on M

and R for c-write algorithms. Theorem 3.13 concerns bounds on W for algorithms that use $\lceil \log k \rceil$ physical registers.

The binary representation algorithm yields an upper bound of $\lceil \log k \rceil$ for R, W, and M. The unary representation algorithm brings down the upper bound for W to 2, while pushing up the bounds for R and M to $\Omega(k)$. This suggests a trade-off between these measures. We can construct a class of algorithms, by borrowing from the two previously mentioned algorithms, that have bounds on R_A and M_A varying from $\Theta(\log k)$ to $\Theta(k)$ and bounds on W_A varying from $\Theta(\log k)$ to $\Theta(1)$.

THEOREM 3.12. *For any m, $1 \leq m \leq k$, there is an algorithm A such that $R_A = \Theta(\log m + k/m)$, $M_A = \Theta(\log m + k/m)$, and $W_A = \lceil \log m \rceil + 2$.*

Proof. We implement our k-ary register by combining an a-ary register and a b-ary register as follows. Let a be the smallest power of 2, which is at least as large as m, i.e., $a = 2^{\lceil \log m \rceil}$. Let $b = \lceil k/a \rceil$. We implement an a-ary register by the *binary* representation method, and a b-ary register by the *unary* representation method. Both these methods have been described earlier. Let the values represented by the a-ary register be in $A = \{1, \ldots, a\}$ and the values represented by the b-ary register be in $B = \{1, \ldots, b\}$. We obtain an ab-ary register by combining these two registers, where the ab values represented are in $A \times B$. Note that $ab \geq k$, so we have our k-ary register.

We consider the bounds of our combination register. The a-ary register uses $\lceil \log m \rceil$ registers and $\lceil \log m \rceil$ physical operations per logical operation. The b-ary register uses $\lceil k/a \rceil$ registers, $\lceil k/a \rceil$ physical reads per logical READ, and 2 physical writes per logical WRITE. Therefore, the total number of registers used is $\lceil \log m \rceil + \lceil k/2^{\lceil \log m \rceil} \rceil$. This is also the number of physical reads per logical READ. There are $\lceil \log m \rceil + 2$ physical writes per logical WRITE. This gives the combined bounds claimed by our theorem. □

The preceding theorem helps us to derive upper bounds for M_S and R_S, where S is the class of c-write algorithms. Choose $m = 2^{c-2}$. Since $c \leq \log k$, it follows that $m \leq k$ and Theorem 3.12 applies. Therefore, there exists an algorithm A such that

(i) $W_A = c$,

(ii) $R_A = c - 2 + \lceil k/2^{c-2} \rceil$, and

(iii) $M_A = c - 2 + \lceil k/2^{c-2} \rceil$.

We thus have the corresponding upper bounds for R_S and M_S, where S is the class of c-write algorithms. Clearly, the upper bounds obtained earlier for the class of 1-write algorithms also hold for c-write algorithms. These new bounds surpass the earlier bounds when $c \geq 3$.

The next theorem states that if an algorithm uses only $\lceil \log k \rceil$ physical registers, then some logical WRITE must use at least $\lceil \log k \rceil$ physical writes.

THEOREM 3.13. *For any algorithm A, if $M_A \leq \lceil \log k \rceil$, then $W_A \geq \lceil \log k \rceil$.*

Proof. Let A be an algorithm with $M_A = \lceil \log k \rceil$. (We have already shown M_A cannot be smaller.) Since the physical registers are binary, $|\mathcal{T}| \leq 2^{\lceil \log k \rceil}$. Recall that for all $v \in V$, there is an $x \in \mathcal{T}$ with $L(x) = v$.

Let U be the subset of $\mathcal{T}$ such that x is in U if and only if there is no $y \neq x$ in $\mathcal{T}$ such that $L(y) = L(x)$. Thus for each configuration x in U, x is the only terminal configuration which has the logical value $L(x)$.

CLAIM 3.14. *There is an $x \in U$ such that $\overline{x} \in \mathcal{T}$. ($\overline{x}$ is the binary string that differs from x in every bit.)*

Proof. Suppose there is no such x. Let $|U| = l$. Each element of U corresponds to a distinct element of V, accounting for l elements of V. The remaining $k - l$ elements of V are represented among the configurations of $\mathcal{T}$ that are not in U and are not the inverse of an element of U. There are at most $2^{\lceil \log k \rceil} - 2l$ of these configurations. There are at least two of these configurations for each remaining element of V. Thus $2^{\lceil \log k \rceil} - 2l \geq 2(k - l)$, which implies $\lceil \log k \rceil \geq \log k + 1$, a contradiction. □

Choose $x \in U$ such that $\overline{x} \in \mathcal{T}$. Let σ be a schedule in $\mathcal{WOC}$ with configuration $\overline{x}$. Suppose $L(x) = v$. Then there is a schedule τ in $\mathcal{WOC}$ of the form σ WRITE(v) α ACK, where α contains no ACK. The configuration of τ must be x since $x \in U$. Thus α contains at least $\lceil \log k \rceil$ writes, and $W_A \geq \lceil \log k \rceil$. □

4. k-ary regular register from binary regular registers. We now shift our attention to regular registers. We would like to implement an n-reader, k-ary, regular register using n-reader, binary, regular registers. Binary regular registers and binary safe registers have the same power. In other words, one can be implemented from the other using one physical register per logical register, at most one physical write per logical WRITE, and one physical read per logical READ [6].

As with safe registers, the problem of implementing k-ary regular registers can also be shown to have a parallel in a combinatorial problem. If there exists an algorithm to implement a k-ary regular register that uses m binary registers, then there is a partial k-coloring of an m-dimensional hypercube with the following restriction. For each colored vertex v, let c be its color. Then, for each color c_i such that $c_i \neq c$ (there are $k-1$ such colors), there exists a path in the hypercube from v to some vertex v_i with color c_i *all of whose intermediate vertices are colored* c.

This characterization takes care of a slow WRITE that overlaps a number of READs. The path corresponds to the intermediate configurations reached during a WRITE. It makes sure that whatever value is RETURNed by a READ that sees an intermediate configuration preserves the regular property of registers. Note, however, that while this restriction is necessary for an algorithm, it is not sufficient. This is because the restriction doesn't take care of the problem of a slow READ overlapping a number of WRITEs, as we will show later. In particular, our hypercube algorithm for safe registers satisfies this characterization, but cannot be used to implement a regular register. Therefore, this characterization may help us get a lower bound for this problem, but not an upper bound.

Subsection 4.1 shows our independent bounds on R, W, and M. Subsection 4.2 contains our trade-off results. As before, we let $V = \{0, \ldots, k-1\}$.

4.1. Independent bounds. The following theorem establishes the independent bounds achieved for this problem.

THEOREM 4.1. *The implementation of an n-reader, k-ary, regular register by n-reader, binary, regular registers gives the following independent bounds*:

(i) $R = \lceil \log k \rceil$,

(ii) $1 \leq W \leq \lceil \log k \rceil$, *and*

(iii) $\max\{\lceil \log k \rceil + 1, \lceil 2 \log k - \log\log k \rceil - 2\} \leq M \leq \min\{k-1, n(3 \log k + 68)\}$.

Proof. The lower bound for R follows directly from a similar proof as the one for safe registers. The lower bound for W is obvious. The lower bound for M is shown in Lemmas 4.4 and 4.5.

The upper bounds on R and W appear simultaneously in the tree algorithm, presented below. However, this algorithm uses $k-1$ physical registers. Lamport [6] describes a complex composition of implementations to achieve an algorithm using $n(3 \log k + 68)$ 1-reader physical registers (recall that n is the number of readers for the logical register). It is unknown whether a better result, for example without the factor of n, is possible by taking advantage of the additional power when the physical registers are n-reader. □

The **modified unary algorithm** is a simple algorithm in [6] that gives upper bounds of $W \leq k$, $R \leq k$ and $M \leq k$. Given registers $X_0, \ldots, X_{k-1}$, the index of the lowest indexed register that has the value 1 determines the k-ary value represented. A READ operation reads $X_0, X_1, \ldots$, in order, until a 1 is returned. It subsequently RETURNs v, where the 1 was read from X_v. A WRITE(v) operation writes 1 in register X_v, and then writes 0 in $X_{v-1}, \dots, X_0$, in order. (It is possible to optimize the algorithm so as to remove register X_{k-1}.)

Our hypercube algorithm, which we used to implement a k-ary safe register from binary safe registers, cannot be used to implement a k-ary regular register. The reason for this is as follows. In case of a slow READ that overlaps a number of WRITEs, the physical reads initiated by the READ may return a set of register values that do not represent a configuration that occurred during the course of the READ. Thus a logical value may be RETURNed that does not correspond to a value written by an overlapping or last preceding WRITE. A stronger result, stating that no 1-write algorithm using $k-1$ registers can implement a k-ary regular register from binary regular registers, is proven in Theorem 4.6.

We now present our new **tree algorithm**, which gives the improved bounds of $R \leq \lceil \log k \rceil$, $W \leq \lceil \log k \rceil$, and $M \leq k-1$. The registers are the nodes in a binary tree. The tree represents a sort of binary search conducted by the READ operation to find the value written. The READ takes a path from the root to a leaf, while the WRITE follows a path starting from a leaf to the root. The path in the tree taken by the READ, along with the values it reads, uniquely defines the value read.

The tree representation of the registers is described as follows. Given any binary tree of k leaves, the internal nodes of the tree correspond to the registers, while the leaves correspond to the k-ary values. Let the leaves of the tree be labeled in some arbitrary manner by the k values in V.

Let v_0 be the initial value of the logical register. The initial values of the physical registers are those that would result from starting with all 0's in the physical registers and then executing a single WRITE(v_0) operation described as follows.

A WRITE(v) operation writes into the set of registers that form the path from the leaf labeled v to the root, beginning with the parent of the leaf, following the path, and ending with the root. The value written to the ith node on the path is 0 (respectively, 1) if the $(i-1)$-st node on the path is the left (respectively, right) child.

A READ operation reads a set of registers that form a path from the root to a leaf labeled v, for some v, beginning with the root. Suppose the ith node read has value 0 (respectively, 1). If its left (respectively, right) child is a leaf, then v is RETURNed, where v is the label of the leaf. Otherwise, the left (respectively, right) child of the ith node is the $(i+1)$-st node read.

We just showed that any binary tree with k leaves completely specifies our algorithm. Simple results in graph theory imply that for any k, there exists a binary tree with k leaves, $k-1$ internal nodes, and height $\lceil \log k \rceil$ (the number of edges in the longest path from the root to a leaf). To obtain the desired complexity bounds, we base our algorithm on one of these trees. Since only internal nodes correspond to registers, $M_A = k-1$, $R_A \leq \lceil \log k \rceil$, and $W_A \leq \lceil \log k \rceil$.

Figure 3 illustrates a 7-ary register with value 3. The path marked on the tree corresponds to the physical registers read by a logical READ operation.

If k is a power of 2, the registers and values form a *complete* binary tree of height $\log k$. We describe the algorithm, for this special case, formally in Fig. 4. Let $v_m v_{m-1} \dots v_1$ be the binary representation of the k-ary value v, where $m = \log k$. The root register is labeled with the empty string ϵ. For each register labeled with the binary string l, the strings $l0$ and $l1$ are the labels of its left and right children, respectively. Let the initial value of the logical register be v_0 with its binary representation being $v_{0,m} v_{0,m-1} \dots v_{0,1}$. Then the initial value of the physical register labeled $v_{0,m} \dots v_{0,p+1}$ is $v_{0,p}$, for all $p \in \{1, \dots, m\}$. All other physical registers have initial value 0.

Here, the $\log k$ physical values read by the READ operation form the binary representation of the k-ary number. Clearly, the algorithm has the bounds stated.

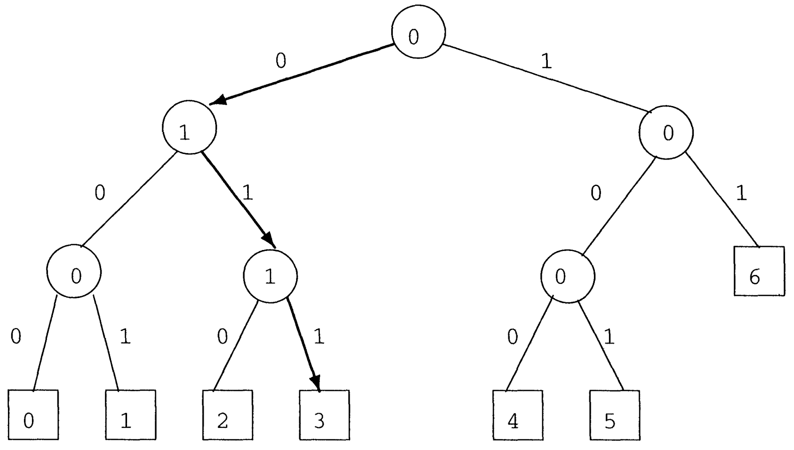

FIG. 3. *An example illustrating the tree algorithm.*

```
To WRITE($v$),
for $p$ := 1 to $m$ do
        write $v_p$ to register $v_m \dots v_{p+1}$
ACK
```

```
To READ,
for $p$ := $m$ to 1 do
        $v_p$ := read register $v_m \dots v_{p+1}$
RETURN($v_m \dots v_1$)
```

FIG. 4. *Tree algorithm for k a power of 2.*

In order to prove the correctness of the tree algorithm, we need some definitions and a lemma. We define a physical read r to **reflect** a physical write w, in a given schedule, if r and w access the same physical register, and either (1) w completely precedes r, or (2) w and r overlap and r returns the value that w writes. We say that a logical READ R **notices** a logical WRITE W if R contains a physical read that reflects a physical write contained in W.

LEMMA 4.2. *Given any schedule of the tree algorithm, and any* READ *R in the schedule, R* RETURNs *the value written by the last* WRITE *W that R notices (note that there is a total order among the* WRITE *operations). If no such* WRITE *exists, R* RETURNs *the initial value.*

Proof. Let R be a READ in some schedule. Suppose R notices no WRITEs. Then every physical read r initiated by R returns the initial value of the physical register read. Therefore, R RETURNs the initial value of the logical register.

Otherwise, R notices some WRITEs. Let W be the last WRITE that R notices. Let s be the last register read by R such that R's read from s reflects W's write to s. Clearly, R reads the value b written by W into s. Otherwise, there is a later WRITE W_1 such that W_1 writes s and R notices W_1; which contradicts the fact that W is the last WRITE that R notices.

Without loss of generality, let $b = 0$. (The argument for $b = 1$ is identical by replacing "left" in the following discussion with "right.")

We claim that s is the last register read by R. Suppose not. Then, R next reads the register t corresponding to the left son of s. Since W wrote b in register s, it must have earlier written to register t. This contradicts the definition of s.

Now, the left son of s must be a leaf node. Let v be the label of this leaf node. Clearly, v is RETURNed by R. Since W writes b into s, the logical value written by W is v. □

THEOREM 4.3. *The tree algorithm implements a k-ary regular register using binary regular registers.*

Proof. We need to argue that our logical k-ary register behaves correctly; i.e., given that our algorithm is implemented using regular binary physical registers, it actually implements a regular k-ary register. Clearly the algorithm has the wait-free property.

Given any schedule, and any READ R in that schedule, we need to prove that R RETURNs the value of one of the WRITE operations it overlaps with or the last preceding WRITE W_1 (or the initial value, in the case that no WRITE completely precedes R). We consider two cases.

Case 1: R notices no WRITEs. Since R reads the root node, and any WRITE must write into the root node, it follows that no WRITE completely precedes R. By Lemma 4.2, R RETURNs the initial value, and this satisfies regularity.

Case 2: R notices some WRITEs. Let W_1 be the last WRITE that R notices. By Lemma 4.2, R RETURNs the value written by W_1. We show that W_1 either overlaps with R or is the last WRITE preceding R. This would satisfy regularity.

Clearly, W_1 cannot completely follow R, since, by the definition of *notice*, W_1 contains a physical write that either precedes or overlaps a physical read contained in R. The only other case to consider is that W_1 precedes another WRITE W_2, which completely precedes R. Since W_1 is the last WRITE that R notices, R does not notice W_2. Since W_2 completely precedes R, R must read the root node after W_2 writes into it, which implies that R does notice W_2. This gives a contradiction. Therefore, W_1 either overlaps with R or is the last WRITE preceding R. □

The tree algorithm simultaneously gives us the best bounds we have for this problem. If the frequencies of READs and WRITEs of all the k values were known in advance, then the number of accesses per READ or WRITE could be optimized by organizing the binary registers as a Huffman tree. For a discussion of Huffman codes, see Hamming [4].

We present our lower bounds for M as follows. Both of the bounds we obtain are significant for different values of k. The bound of Lemma 4.5 supersedes the bound of Lemma 4.4 for $k \geq 55$.

LEMMA 4.4. $M \geq \lceil \log k \rceil + 1$.

Proof. Choose any algorithm A. We assume, for contradiction, that $M_A = \lceil \log k \rceil$. Note that the lower bound for M of $\lceil \log k \rceil$, proved for safe registers, holds here as well. For all $v \in V$, there is a schedule σ_v of A in $\mathcal{WOC}$ of the form WRITE(v) α_v ACK, where α_v contains no ACK. Let C_v be the configuration of σ_v; it is easy to see that C_v is stable.

Choose $v \in V$. For each $w \in V$, $w \neq v$, there is a schedule σ_{vw} in $\mathcal{WOC}$ of the form WRITE(v) α_v ACK WRITE(w) β_{vw} ACK, where β_{vw} contains no ACK. Let C_{vw} be the configuration of σ_{vw}; it is easy to see that C_{vw} is stable.

Since only WP takes steps in σ_{vw} and physical writes are done serially, β_{vw} goes through a sequence of stable configurations (corresponding to schedules in $\mathcal{WO}$). By Lemma 2.2, $L_1(C_{vw}) = w$ and $L_1(C_v) = v$. Since $w \neq v$ and L_1 is a function by Lemma 2.1, $C_{vw} \neq C_v$. Thus a stable configuration is reached in β_{vw} that is different than C_v. Let D_{vw} be the first such configuration. D_{vw} and C_v differ in a single bit, i.e., in the value of a single register.

Since there are only $\lceil \log k \rceil$ bits in each configuration, there are only $\lceil \log k \rceil$ configurations that differ in a single bit from C_v. Since there are $k - 1$ values in V different than v, there exist distinct w and u in V such that $D_{vw} = D_{vu}$. Call this configuration D_v. By regularity, $L_1(D_{vw}) \in \{v, w\}$ and $L_1(D_{vu}) \in \{v, u\}$. Thus $L_1(D_v) = v$.

Since $L_1(C_v) = v$, all the C_v's are distinct. Since $L_1(D_v) = v$, all the D_v's are distinct. It is easy to see that $C_v \neq D_w$ for all v and w. Thus there are at least $2k$ distinct stable configurations, requiring at least $\lceil \log k \rceil + 1$ registers. Therefore, we have a contradiction. □

LEMMA 4.5. $M \geq \lceil 2 \log k - \log\log k \rceil - 2$.

Proof. Choose any algorithm A. Let d be the number of registers used in the algorithm.

For all $v \in V$, there is a schedule σ_v of A in $\mathcal{WOC}$ of the form WRITE(v) α_v ACK, where α_v contains no ACK. Let C_v be the configuration of σ_v; it is easy to see that C_v is stable. Clearly, $L_1(C_v) = v$.

We claim that for any two k-ary values v and w, there exists a pair of stable configurations D_v and D_w that differ in exactly one bit such that $L(D_v) = v$ and $L(D_w) = w$. Suppose not. Then, consider the schedule σ_{vw} in $\mathcal{WOC}$ of the form σ_v WRITE(w) β_{vw} ACK, where β_{vw} contains no ACK. Let the configuration of σ_{vw} be D_{vw}. The configuration of σ_v is C_v. Note that D_{vw} is a stable configuration and $L_1(D_{vw}) = w$. Consider the sequence of stable configurations reached by the schedule σ_{vw} starting from C_v and ending at D_{vw}. By the assumption, there exists a stable configuration D_x in the sequence such that $L_1(D_x) = x$ but $x \neq v$ and $x \neq w$. A READ starting at D_x would therefore RETURN x, which violates regularity. This gives a contradiction.

Recall that $\mathcal{S}$ is the set of all configurations resulting from schedules in $\mathcal{WO}$ (only WP takes steps and no physical write is pending). Let c_v be the number of stable configurations C in $\mathcal{S}$ such that $L_1(C) = v$, for each k-ary value v. Let $c = \min\{c_x | x \in V\}$, and let $v \in V$ be such that $c = c_v$. For each value w such that $w \neq v$, there are stable configurations D_v and D_w in $\mathcal{S}$ that differ in exactly one bit such that $L_1(D_v) = v$ and $L_1(D_w) = w$. Since each stable configuration C, such that $L_1(C) = v$, has d neighbors, and there are $(k-1)$ values w, it follows that $cd \geq k-1$. Since there are k different values and at most 2^d possible stable configurations, $ck \leq 2^d$. Solving the two inequalities, we obtain that $k^2 - k \leq d\,2^d$, which implies that $2(\log k) \leq d + \log d + 1$, for $k \geq 2$. The last inequality implies that $d \geq 2(\log k) - \log\log k - 2$. □

4.2. Trade-offs. We have the following lower bounds for R and M relating to one-write algorithms. In particular, we show that any one-write algorithm for this problem would require at least k registers. In other words, our hypercube algorithm for safe registers does not work for regular registers.

THEOREM 4.6. *For all algorithms A, if $W_A = 1$ then $R_A \geq k-1$ and $M_A \geq k$.*

Proof. The lower bound for R_A follows from a similar proof as the one for safe registers. By using a similar argument, we can actually make the additional claim that every READ reads at least $k-1$ distinct physical registers. We use this claim in the following proof of the bound for M_A.

To show $M_A \geq k$, suppose in contradiction that a one-write algorithm A exists that uses $k-1$ registers. Then Lemma 3.6 carries over from the safe case, implying that the function L has the rainbow-coloring property. Let C_0 be the initial configuration; clearly, $L(C_0) = v_0$. Consider the following schedule α: READ(1) δ RETURN(1, v_0), where δ consists only of physical actions taken by RP$_1$. We claim that δ does not contain any physical write.

CLAIM 4.7. *The sequence of actions δ does not contain a physical write.*

Proof. Suppose δ does contain a physical write, i.e., $\delta = \delta_1$ write$_i(b)$ δ_2, where δ_1 contains no physical write. Then, there is a schedule of the form

$$\text{READ}(1)\ \delta_1\ \text{write}_i(b)\ \delta_2\ \text{RETURN}(1, v_0)\ \text{READ}(1)\ \delta'\ \text{RETURN}(1, v_0),$$

where δ' contains only physical actions. Let C_1 be the configuration that differs from C_0 only in position i. Then $L(C_1) = v$, for some $v \neq v_0$.

Consider the schedule WRITE(v) γ ACK, where γ contains only physical actions of WP. Then γ consists of a single physical write, to register i (as well as possibly some physical reads). An easy induction shows that

$$\text{READ}(1)\ \delta_1\ \text{WRITE}(v)\ \gamma\ \text{ACK write}_i(b)\ \delta_2\ \text{RETURN}(1, v_0)\ \text{READ}(1)\ \delta'\ \text{RETURN}(1, v_0)$$

is a schedule, since there is no physical write in δ_1 and the physical write within the logical

WRITE is "obliterated" by write$_i(b)$. This violates regularity because the second READ should RETURN v, not v_0. □

Now, we continue with the proof of the theorem. Pick two distinct registers (call them registers i and j) that are read in schedule α.

We define C_1 to be the stable configuration that differs from C_0 in position j, C_3 to be the stable configuration that differs from C_0 in position i, and C_2 to be the stable configuration that differs from C_0 in positions i and j. For all $l \in \{1, 2, 3\}$, C_l is a terminal configuration. Let $L(C_l) = v_l$. It is easy to verify that v_0, v_1, v_2, and v_3 are distinct values in V. Suppose, without loss of generality, that the initial value of both registers i and j is 0. Figure 5 illustrates the relation between the four configurations defined. Adjacent configurations differ in a single bit. The label on the edge between two configurations corresponds to the particular bit in which they differ.

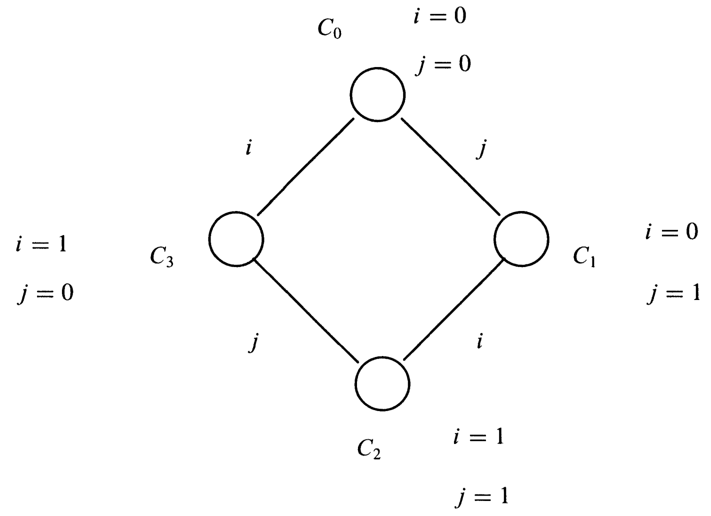

FIG. 5. *Relationship between the four configurations.*

Now, consider the sequences of actions, specified in Table 3, which can be applied at a configuration C_{start} and results in the configuration C_{finish}.

TABLE 3
Sequences for proof of Theorem 4.6.

C_{start}	sequence β	C_{finish}
C_0	β_{01} = WRITE(v_1) γ_{01} write$_j$(1)γ'_{01} ACK	C_1
C_1	β_{12} = WRITE(v_2) γ_{12} write$_i$(1)γ'_{12} ACK	C_2
C_2	β_{23} = WRITE(v_3) γ_{23} write$_j$(0)γ'_{23} ACK	C_3
C_3	β_{32} = WRITE(v_2) γ_{32} write$_j$(1)γ'_{32} ACK	C_2
C_2	β_{21} = WRITE(v_1) γ_{21} write$_i$(0)γ'_{21} ACK	C_1

We claim that if we have a schedule σ with the configuration C_{start} and no pending WRITE, we can concatenate an appropriate sequence of actions β (from Table 3) to σ to obtain the schedule σ' with the configuration C_{finish}. The sequence β is a single logical WRITE which consists of a single physical write (and possibly some physical reads)—thus none of the γ_{ab}'s contain any physical writes. It is easy to see that each β exists.

We create a new sequence α' by taking α and inserting certain sequences at certain points, according to the following rules. First, we insert β_{01} immediately before READ(1), resulting in configuration C_1. Then, immediately before each read$_j$ of RP$_1$, if the configuration is C_1, we insert $\beta_{12}\beta_{23}$, resulting in configuration C_3. Immediately before each read$_i$ of RP$_1$, if the configuration is C_3, we insert $\beta_{32}\beta_{21}$, resulting in configuration C_1.

To see that α' is a schedule, it is sufficient to observe that the only time the configuration changes within the schedule is when a sequence β_{ab} is inserted. This follows from the fact, proven in Claim 4.7, that α contains no physical writes. In particular, inserting β_{01} changes the configuration to C_1, inserting $\beta_{12}\beta_{23}$ changes the configuration to C_3, and inserting $\beta_{32}\beta_{21}$ changes the configuration to C_1. We can prove, by a simple induction, that the configuration reached by any prefix of schedule α' up to a read$_i$ by RP$_1$ is always C_1. Similarly, the configuration reached by any prefix of schedule α' up to a read$_j$ by RP$_1$ is always C_3. Therefore, read$_i$ and read$_j$ always return the value 0. It follows that v_0 is the value RETURNed by the READ(1) in the schedule α'. Since, to satisfy regularity, the READ should RETURN v_1, v_2, or v_3, we have a contradiction. □

We conclude this section with a trade-off result relating to a constant number of writes. This follows from the identical result derived in the safe case.

THEOREM 4.8. *For all algorithms A, if $W_A = c$, where $c \leq (\log k)/3$, then $M_A \geq (c!k/2)^{1/c}$ and $R_A \geq (c!k/2)^{1/c}$.*

5. Conclusion. We have demonstrated upper and lower bounds on the number of physical registers, the number of physical reads in a logical read, and the number of physical writes in a logical write, for a variety of multivalued register implementations. In many cases, our bounds are tight. Some of our upper bounds follow from two new algorithms that we present, one for implementing a k-ary safe register out of binary safe registers, and another for implementing a k-ary regular register out of binary regular registers. We also presented several interesting trade-offs between these cost measures, for implementing k-ary registers out of binary registers. The bounds on the number of physical operations can be converted into bounds on the time to perform the logical operations, in terms of the time for the physical operations.

Future work includes finding such bounds for more algorithms, in particular, those involving atomic registers and multi-writer registers. The bounds in this paper on W and M for implementing a k-ary regular register out of binary regular registers are not tight. (Current work shows that the tight bound for W is 1, i.e., that there exists a 1-write algorithm for a k-ary regular register [3].) A final question is what difference does it make, if any, if clocks are available to the read and write processes?

Acknowledgments. We would like to thank Brian Coan for the proof of Lemma 3.7. Akhilesh Tyagi suggested the combinatorial techniques required in proving Theorems 3.10 and 3.11. We are grateful to Hagit Attiya, Paul Beame, Richard Ladner, Mike Saks, Martin Tompa, and George Welch for many helpful conversations that improved some of the results and presentation. The notes [8] from the Distributed Algorithms class taught by Nancy Lynch at MIT served as a useful survey and tutorial of earlier work. The anonymous referees made helpful comments that improved the presentation.

REFERENCES

[1] B. BLOOM, *Constructing two-writer atomic registers*, in Proceedings of the Sixth Annual Association for Computing Machinery SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Vancouver, Canada, August 1987, pp. 249–259.

[2] J. E. Burns and G. L. Peterson, *Constructing multi-reader atomic values from non-atomic values*, in Proceedings of the Sixth Annual Association for Computing Machinery SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Vancouver, Canada, August 1987, pp. 222–231.

[3] S. Chaudhuri, M. Kosa, and J. L. Welch, *Upper and lower bounds for one-write multivalued regular registers*, in Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing, Dallas, December 1991, pp. 134–141.

[4] R. W. Hamming, *Coding and Information Theory*, 2nd ed., Prentice Hall, Englewood Cliffs, NJ, 1986.

[5] G. Kant and J. van Leeuwen, *The file distribution problem for processor networks*, in Lecture Notes in Computer Science 447: Proceedings of the Second Scandinavian Workshop on Algorithm Theory, Springer-Verlag, Berlin, 1990, pp. 48–59.

[6] L. Lamport, *On interprocess communication*, Distributed Computing, 1 (1986), pp. 86–101.

[7] M. Li, J. Tromp, and P. M. B. Vitanyi, *How to Share Concurrent Wait-Free Variables*, manuscript.

[8] N. A. Lynch and K. J. Goldman, *Distributed Algorithms: Lecture Notes for* 6.852, Research Seminar Series MIT/LCS/RSS 5, Massachusetts Institute of Technology, Cambridge, MA, May 1989.

[9] N. A. Lynch and M. R. Tuttle, *Hierarchical correctness proofs for distributed algorithms*, in Proceedings of the Sixth Annual Association for Computing Machinery SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Vancouver, Canada, August 1987, pp. 137–151.

[10] R. Newman-Wolfe, *A protocol for wait-free, atomic, multi-reader shared variables*, in Proceedings of the Sixth Annual Association for Computing Machinery SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Vancouver, Canada, August 1987, pp. 232–248.

[11] G. Peterson, *Concurrent reading while writing*, Association for Computing Machinery Trans. Programming Languages and Systems, 5 (1983), pp. 46–55.

[12] A. K. Singh, J. H. Anderson, and M. G. Gouda, *The elusive atomic register revisited*, in Proceedings of the Sixth Annual Association for Computing Machinery SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Vancouver, Canada, August 1987, pp. 206–221.

[13] J. T. Tromp, *How to Construct an Atomic Variable*, Tech. report CS-R8939, Centre for Mathematics and Computer Science, Amsterdam, the Netherlands, October 1989.

[14] A. Tyagi, *The Role of Energy in VLSI Computations*, Ph.D. thesis, Department of Computer Science, University of Washington, Seattle, WA, 1988. Available as UWCS Tech. report 88-06-05.

[15] K. Vidyasankar, *Converting Lamport's regular register to atomic register*, Inform. Process. Lett., 28 (1988), pp. 287–290.

[16] ———, *Concurrent reading while writing revisited*, Distributed Computing, 4 (1990), pp. 81–85.

[17] P. M. B. Vitanyi and B. Awerbuch, *Atomic shared register access by asynchronous hardware*, in Proceedings of the Twenty-Seventh Annual IEEE Symposium on Foundations of Computer Science, Toronto, Canada, October 1986, pp. 233–243.