A

Polylogarithmic Concurrent Data Structures from Monotone Circuits

JAMES ASPNES, Yale University
HAGIT ATTIYA, Technion
KEREN CENSOR-HILLEL, MIT

The paper presents constructions of useful concurrent data structures, including max registers and counters, with step complexity that is sublinear in the number of processes, n. This result avoids a well-known lower bound by having step complexity that is polylogarithmic in the number of values the object can take or the number of operations applied to it.

The key step in these implementations is a method for constructing a *max register*, a linearizable, wait-free concurrent data structure that supports a write operation and a read operation that returns the largest value previously written. For fixed m, an m-valued max register is constructed from one-bit multi-writer multi-reader registers at a cost of at most $\lceil \log m \rceil$ atomic register operations per write or read. An *unbounded* max register is constructed with cost $O(\min(\log v, n))$ to read or write a value v.

Max registers are used to transform any monotone circuit into a wait-free concurrent data structure that provides write operations setting the inputs to the circuit and a read operation that returns the value of the circuit on the largest input values previously supplied. One application is a simple, linearizable, wait-free counter with a cost of $O(\min(\log n \log v, n))$ to perform an increment and $O(\min(\log v, n))$ to perform a read, where v is the current value of the counter. For polynomially-many increments, this becomes $O(\log^2 n)$, an exponential improvement on the best previously known upper bounds of $O(n)$ for exact counting and $O(n^{4/5+\epsilon})$ for approximate counting.

Finally, it is shown that the upper bounds are almost optimal. It is shown that for deterministic implementations, even if they are only required to satisfy solo-termination, $\min(\lceil \log m \rceil, n-1)$ is a lower bound on the worst-case complexity for an m-valued bounded max register, which is exactly equal to the upper bound for $m \leq 2^{n-1}$, and $\min(n-1, \lceil \log m \rceil - \log(\lceil \log m \rceil + k))$ is a lower bound for the read operation of an m-valued k-additive-accurate counter, which is a bounded counter in which a read operation is allowed to return a value within an additive error of $\pm k$ of the number of increment operations linearized before it. Furthermore, even in a solo-terminating randomized implementation of an n-valued max register with an oblivious adversary and global coins, there exist simple schedules in which, with high probability, the worst-case step complexity of a read operation is $\Omega(\log n / \log \log n)$ if the write operations have polylogarithmic step complexity.

Categories and Subject Descriptors: D.1.3 [**Software**]: Programming Techniques—*Concurrent programming*; E.1 [**Data**]: Data Structures; F.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity—*Nonnumerical Algorithms and Problems*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: distributed computing, shared memory, max registers, counters, monotone circuits

---

A preliminary version of this paper appeared in Proceedings of the *28th Annual ACM Symposium on Principles of Distributed Computing*, pages 36-45, 2009. James Aspnes is partially supported by NSF grant CNS-0435201. Hagit Attiya is partially supported by the *Israel Science Foundation* (grant number 953/06). Keren Censor-Hillel is supported by the Simons Postdoctoral Fellows Program. Part of this author's work was done as a Ph.D. student at the Department of Computer Science, Technion, and supported in part by the Adams Fellowship Program of the Israel Academy of Sciences and Humanities.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

ACM Reference Format:
J. ACM V, N, Article A (January YYYY), 25 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

A critical aspect of programming contemporary multiprocessing systems is the implementation of *concurrent data structures*, e.g., getting the largest value stored in a data structure, or counting. It is important to find methods for building efficient concurrent data structures in shared-memory systems, where n processes communicate by reading and writing to shared *multi-writer multi-reader* registers.

One successful approach to building concurrent data structures is to employ the *atomic snapshot* abstraction [Afek et al. 1993; Anderson 1993; Aspnes and Herlihy 1990]. An atomic snapshot object is composed of components, each of which typically updated by a different processes; the components can be atomically read. By applying a specific function to the components read, we can provide a specific data structure. For example, to obtain a *max register*, supporting a write operation and a `ReadMax` operation that returns the largest value previously written, the function returns the component with the maximum value; to obtain a *counter*, supporting an increment operation and a `ReadCounter` operation, the function adds up the contribution from each process.

These constructions take a linear (in n) number of steps, due to the cost of implementing atomic snapshots [Inoue and Chen 1994]. Indeed, Jayanti et al. [2000] show that operations must take $\Omega(n)$ space and $\Omega(n)$ steps in the worst case, for many common data structures, including max registers and counters. This seems to indicate that we cannot do better than snapshots.

However, careful inspection of Jayanti, Tan, and Toueg's lower bound proof reveals that it holds only when there are numerous operations on the data structure. Thus, it does not rule out the possibility of having sub-linear algorithms when the number of operations is bounded, or, more generally, the existence of algorithms whose complexity depends on the number of operations. Such data structures are useful for many applications, either because they have a limited life-time, or because several instances of the data structure can be used.

Our Contributions. In this paper, we present polylogarithmic implementations of key data structures that support only bounded values. The cornerstone of our constructions, and our first example, is an implementation of a max register that beats the $\Omega(n)$ lower bound of Jayanti et al. [2000] when $\log m$ is $o(n)$.[1] If the number of values is bounded by m, its cost per operation is $O(\log m)$; for an unbounded set of values, the cost is $O(\min(\log v, n))$, where v is the value of the register.

To implement a counter, instead of simply summing the individual process contributions, as in a snapshot-based implementation of a counter, we can use a tree of max registers to compute this sum: take an $O(\log n)$ depth tree of two-input adders, where the output of each adder is a max register. To increment, walk up the tree recomputing all values on path. The cost of a read operation is $O(\min(\log v, n))$, where v is the current value of counter, and the cost of an increment operation is $O(\min(\log n \log v, n))$. When the number of increments is polynomial, this has $O(\log^2 n)$ cost, which is an exponential improvement from the trivial upper bound of $O(n)$ using snapshots. The resulting counter is wait-free and linearizable.

More generally, we show how a max register can be used to transform any monotone circuit into a wait-free concurrent data structure that provides write operations set-

[1] Throughout the paper we use $\log$ to denote $\log_2$.

ting the inputs to the circuit and a read operation that returns the value of the circuit on the largest input values previously supplied. Monotone circuits expose the parallelism inherent in the dependency of the data structure's values on the arguments to the operations. Formally, a *monotone circuit* computes a function over some finite alphabet of size m, which is assumed to be totally ordered. The circuit is represented by a directed acyclic graph where each node corresponds to a gate that computes a function of the outputs of its predecessors. Nodes with in-degree zero are input nodes; nodes with out-degree zero are output nodes. Each gate g, with k inputs, computes some monotone function f_g of its inputs. Monotonicity means that if $x_i \geq y_i$ for all i, then $f_g(x_1, \ldots, x_k) \geq f_g(y_1, \ldots, y_k)$.

The cost of writing a new value to an input to the circuit is bounded by $O(Sd \min(\lceil \log m \rceil, n)$, where m is the size of the alphabet for the circuit, d is the number of inputs to each gate, and S is the number of gates whose value changes as the result of the write. The cost of reading the output value is $\min(\lceil \log m \rceil, O(n))$. While the resulting data structure is not linearizable in general, it satisfies a weaker but natural consistency condition, called *monotone consistency*, which we will show is still useful for many applications.

So far we have only described upper bounds. We show a lower bound of $\min(\lceil \log m \rceil, n-1)$ steps for the read operation of any solo-terminating deterministic implementation of a max register when $m \leq n$. This implies that our implementation is optimal. A bound of $\min(n-1, \lceil \log m \rceil - \log(\lceil \log m \rceil + k))$ steps is shown to hold for the read operation of any solo-terminating deterministic implementation of an m-valued k-additive-accurate counter, which is a bounded counter in which a read operation is allowed to return a value within an additive error of $\pm k$ of the number of increment operations linearized before it.

Furthermore, we show a lower bound for randomized implementations of a max register which exhibits a tradeoff between the number of steps required for read and write operations. It is shown that in any randomized implementation of an n-valued max register, there exist simple schedules containing $n-1$ partial write operations and one read operation in which, with probability $1 - o(1)$, one of the following holds:

(1) The write operation with maximum value takes more than w steps.
(2) The read operation returns an incorrect value.
(3) The read operation takes $\Omega(\log n / (\log w + \log \log n))$ steps.

This applies for an oblivious adversary, which determines the entire schedule in advance, even when requiring only solo-termination, and allowing global coins, which are random values shared by all processes (as opposed to local coins). In particular, this tradeoff shows an $\Omega(\log n / \log \log n)$ lower bound on the worst-case step complexity of read operations for any randomized max register whose write operations have polylogarithmic step complexity.

Related Work. Previously, the linear lower bound of Jayanti et al. [2000] motivated, e.g., Aspnes and Censor [2009] to switch to approximate counting, but even this implementation has $O(n^{4/5+\epsilon})$ cost. Our construction improves this for the case of bounded counters.

Regarding snapshot-based implementations, another approach that improves upon the snapshot-based implementation of many data structures, is to use *f-arrays*, as proposed by Jayanti [2002]. An f-array is a data structure that supports computation of a function f over the components of an array. Instead of having a process take a snapshot of the array and then locally apply f to the result, Jayanti implements an f-array by having the write operations update a predetermined location according to the new value of f, which requires a read operation to only read that location. This

construction is then extended to a tree algorithm. For implementing an f-array of m registers, where f can be any common aggregate function, including the maximum value or the sum of values, this reduces the number of steps required to $O(\log m)$ for a write operation, while a read operation takes $O(1)$ steps. These implementations use base objects that are stronger than read / write registers (specifically, LL/SC), while we restrict our base objects to multi-writer multi-reader registers. Hence, we do not use this approach in this paper.

Model. We assume the standard model of an asynchronous shared-memory system (cf. [Attiya and Welch 2004]), which consists of a set of n processes $P = \{p_0, \ldots, p_{n-1}\}$. Processes communicate by reading and writing to shared multi-writer multi-reader registers.

Each *step* consists of some local computation and one shared memory event, which is either a read or a write to some register. The *schedule* according to which processes take steps is determined by an *adversary*. The system is *asynchronous*, which implies that there are no timing assumptions, and specifically no bounds on the time between two steps of a process, or between steps of different processes.

We are interested in *wait-free* implementations, in which any operation on the data structure terminates within a finite number of its steps regardless of the schedule chosen by the adversary (even if all other processes stop taking steps). The cost of an implementation is measured by the maximum number of steps required for any operation on the data structure. Our lower bounds hold even for the weaker *solo-termination* condition, in which an operation is only required to terminate within a finite number of its steps if there are no concurrent steps by operations of other processes.

An implementation should also satisfy some *consistency condition*. The consistency condition should specify the semantic requirements of all possible executions, including possibly both *complete operations*, which start and terminate during the execution, and *incomplete operations*, which begin during the execution but still have low-level shared memory accesses pending. While the *sequential semantics* of a data structure has to address only non-concurrent operations, our data structures are concurrent, which means that during an execution some operations may overlap, in which case their shared-memory accesses are interleaved. Operations whose shared memory accesses are not interleaved are called *non-overlapping*. Since a consistency condition has to specify the semantic requirements for all possible concurrent executions, it also has to address the possibility of overlapping operations.

Some of our implementations are *linearizable* [Herlihy and Wing 1990]. This means that for any execution, there is a sequence that contains all the completed operations, as well as some of the incomplete ones, that

(1) extends the order of non-overlapping operations; and
(2) preserves the sequential semantics of the implemented object.

Other implementations are not linearizable, but satisfy what we call *monotone consistency*, a weaker yet natural consistency condition defined formally in Section 4.

In a randomized implementation, a process is allowed to flip coins as part of its local computation, and choose its next step according to their results. We require an operation of a randomized implementation of a data structure to be correct with high probability, i.e., with probability $1 - o(1)$. The step complexity of an operation in a randomized implementation is an upper bound on the number of steps required for that operation with high probability. This implies that in a randomized implementation with probability $1 - o(1)$ every operation returns the correct value within its step complexity.

2. MAX REGISTERS

Our basic data structure is a *max register*, which is an object r that supports a $\texttt{WriteMax}(r, t)$ operation with an argument t that records the value t in r, and a $\texttt{ReadMax}(r)$ operation returning the maximum value written to the object r. A max register may be either bounded or unbounded. For a bounded max register, we assume that the values it stores are in the range $\{0, \ldots, (m-1)\}$, where m is the *size* of the register. We assume that any non-negative integer can be stored in an unbounded max register. In general, we will be interested in unbounded max registers, but will consider bounded max registers in some of our constructions and lower bounds.

One way to implement max registers is by using snapshots. Given a linear-time snapshot protocol (e.g., [Inoue and Chen 1994]), a `WriteMax` operation for process p_i updates location $a[i]$, while a `ReadMax` operation takes a snapshot of all locations and returns the maximum value. Assuming no bounds on the size of snapshot array elements, this gives an implementation of an unbounded max register with linear cost (in the number of processes n) for both `WriteMax` and `ReadMax`. We show below how to build more efficient max registers: a recursive construction that gives costs logarithmic in the size of the register for both `WriteMax` and `ReadMax`. We also describe (in Section 6.4) a non-linearizable, Monte Carlo implementation with only one atomic register read per `ReadMax`, but the cost of `WriteMax` is drastically increased; while impractical, it is useful for illustrating the limitations of our lower bounds.

In the remainder of this section we show how to construct a max register recursively from a tree of increasingly large max registers. Each node in this tree is a one-bit register, that directs the reader left or right depend on its value. (See Figure 1.)

Intuitively, we can think of a read of the max register as following a path through this tree, constructing the sequence of bits found in the registers along the path and then treating this sequence as an element of a prefix-free code that can be decoded to obtain the actual value. A write operation similarly walks down along the path determined by the encoding of its input, testing to see if a larger value has already been written. If it finds one (because it sees a 1 in a register where its bit-vector would put a 0), it exits without changing any of the bits in the tree; this mechanism prevents out-of-date writes from entering non-linearizable values that would otherwise be observed by out-of-date reads. But if the writer successfully reaches a leaf of the tree without seeing evidence of a larger value, it sets the 1 bits on its path needed to cause a reader to follow it in bottom-up order. This ordering ensures that no reader is diverted to the new path until it has been completely marked.

For convenience of both implementation and the correctness proof, it is easiest to describe this process recursively, where we think of a max register as built from a one-bit atomic register switching between two smaller max registers. The smallest such registers are trivial $\texttt{MaxReg}_0$ objects, which are max registers r with size $m = 1$ that support only the value 0. The implementation of $\texttt{MaxReg}_0$ requires zero space and zero step complexity: $\texttt{WriteMax}(r, 0)$ is a no-op, and $\texttt{ReadMax}(r)$ always returns 0.

To get larger max registers, we combine smaller ones recursively (see Figure 1). The base objects will consist of at most one snapshot-based max register as described earlier (used to limit the depth of the tree in the unbounded construction) and a large number of trivial $\texttt{MaxReg}_0$ objects and read/write registers. A recursive `MaxReg` object has three components: two `MaxReg` objects called r.left and r.right, where r.left is a bounded max register of size m, and one 1-bit multi-writer register called r.switch. The resulting object is a max register whose size is the sum of the sizes of r.left and r.right, or unbounded if r.right is unbounded.

Pseudocode for the max-register implementation is given in Algorithm 1. Writing a value t to r is done using the $\texttt{WriteMax}(r, t)$ procedure. For values t less than m,

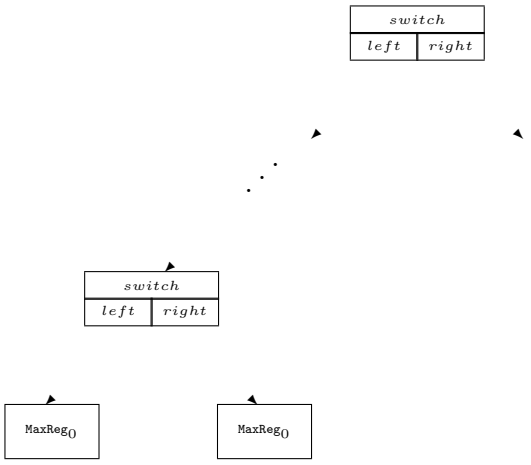

Fig. 1. Implementing a max register.

a process first checks that r.switch is off, and only then writes t to r.left. For larger values, the process writes $t - m$ to r.right and then sets r.switch. Reading the maximal value is done using the `ReadMax`(r) procedure. Here the process uses r.switch to choose whether to read r.left or r.right; if it reads r.right, it adds m to the result.

```
   shared data: switch: a 1-bit multi-writer register, initially 0
                left, a MaxReg object of size m, initially 0,
                right, a MaxReg object of arbitrary size, initially 0
 1 procedure WriteMax(r, t)
 2 begin
 3   if t < m then
 4     if r.switch = 0 then
 5       WriteMax(r.left, t)
 6     end
 7   else
 8     WriteMax(r.right, t − m)
 9     r.switch ← 1
10   end
11 end
12 procedure ReadMax(r)
13 begin
14   if r.switch = 0 then
15     return ReadMax(r.left)
16   else
17     return ReadMax(r.right) + m
18   end
19 end
```

Algorithm 1: Max register implementation

An important property of this implementation is that it preserves linearizability, as shown in the following lemma.

LEMMA 2.1. *If* r.left *and* r.right *are linearizable max registers, so is* r.

PROOF. We assume that each of the MaxReg objects r.left and r.right is linearizable. Thus, we can associate each operation on them with one linearization point and treat these operations as atomic. In addition, we can associate each read or write to the register r.switch with a single linearization point since it is atomic.

We now consider a schedule of $\texttt{ReadMax}(r)$ and $\texttt{WriteMax}(r,t)$ operations. These consist of reads and writes to r.switch and of ReadMax and WriteMax operations on r.left and r.right. We divide the operations on r into three categories:

— C_{left}: $\texttt{ReadMax}(r)$ operations that read 0 from r.switch, and $\texttt{WriteMax}(r,t)$ operations with $t < m$ that read 0 from r.switch.
— C_{right}: $\texttt{ReadMax}(r)$ operations that read 1 from r.switch, and $\texttt{WriteMax}(r,t)$ operations with $t \geq m$ (i.e., that write 1 to r.switch).
— C_{switch}: $\texttt{WriteMax}(r,t)$ operations with $t < m$ that read 1 from r.switch.

Inspection of the code shows that each completed operation on r falls into exactly one of these categories. Notice that an operation is in C_{left} if and only if it invokes an operation on r.left, an operation is in C_{right} if and only if it invokes an operation on r.right, and an operation is in C_{switch} if and only if it invokes no operation on r.left or r.right. We order the operations by the following four rules:

(1) We order all operations of C_{left} before those of C_{right}.
(2) An operation op in C_{switch} is ordered at the latest point possible before any operation op' that starts after op finishes. If two operations are ordered in the same point, then they appear according to the order they are encountered.
(3) Within C_{left} we order the operations according to the order at which they access r.left, i.e., by the order of their respective linearization points in accessing r.left.
(4) Within C_{right} we order the operations according to the order at which they access r.right, i.e., by the order of their respective linearization points in accessing r.right).

It is easy to verify that these rules are well-defined.

We first prove that these rules preserve the execution order of non-overlapping operations. For two operations in the same category this is clearly implied by rules 2–4. Since rule 1 shows that two operations from C_{left} and C_{right} are also properly ordered because an operation that starts after an operation in C_{right} finishes cannot be in C_{left}, it is left to consider the case that one operation is in C_{switch} and the other is either in C_{left} or in C_{right}. In this case, rule 2 implies that their order preserves the execution order.

We now prove that this order satisfies the specification of a max register, i.e., if a $\texttt{ReadMax}(r)$ operation op returns t then t is the largest value written by operations on r of type WriteMax that are ordered before op. This requires showing that there is a $\texttt{WriteMax}(r,t)$ operation op_w ordered before op, and that there is no $\texttt{WriteMax}(r,t')$ operation $op_{w'}$ with $t' > t$ ordered before op.

This is obtained by using the linearizability of the components. If op returns a value $t < m$ (i.e., it is in C_{left}) then this is the value that is returned from its invocation op' of $\texttt{ReadMax}(r.\text{left})$. By the linearizability of r.left, there is a $\texttt{WriteMax}(r.\text{left},t)$ operation op'_w ordered before op' in the linearization of r.left. By rule 3, this implies that the $\texttt{WriteMax}(r,t)$ operation op_w which invoked op'_w is ordered before op. A similar argument for r.right applies if op returns a value $t \geq m$.

To prove that no operation of type WriteMax with a larger value is ordered before op, we assume, towards a contradiction, that there is a $\texttt{WriteMax}(r,t')$ operation $op_{w'}$ with $t' > t$ that is ordered before op. If op returns a value $t < m$ (i.e., it is in C_{left}) then $op_{w'}$ cannot be in C_{right}, otherwise it would be ordered after op, by rule 1. Moreover, $op_{w'}$ cannot be in C_{switch}, since rule 2 implies that op starts after $op_{w'}$ finishes and

hence must also read 1 from r.switch which is a contradiction to $op \in C_{\mathsf{left}}$. Therefore, $op_w \in C_{\mathsf{left}}$, but this contradicts the linearizability of r.left. If op returns a value $t \geq m$ (i.e., it is in C_{right}) then $op_{w'}$ cannot be in C_{left} because $t' > t$. Moreover, $op_{w'}$ cannot be in C_{switch}, since $t' > t \geq m$. Therefore, op_w is in C_{right}, which contradicts the linearizability of r.right. □

Using Lemma 2.1, we can build a max register whose structure corresponds to an arbitrary binary tree, where each internal node of the tree is represented by a recursive max register and each leaf is a $\texttt{MaxReg}_0$, or, for the rightmost leaf, a $\texttt{MaxReg}_0$ or snapshot-based `MaxReg` depending on whether we want a bounded or an unbounded max register. There are several natural choices, as we will discuss next.

2.1. Using a balanced binary tree

To construct a bounded max register of size 2^k, we use a balanced binary tree. Let $\texttt{MaxReg}_k$ be a recursive max register built from two $\texttt{MaxReg}_{k-1}$ objects, with $\texttt{MaxReg}_0$ being the trivial max register defined previously. Then $\texttt{MaxReg}_k$ has size 2^k for all k. It is linearizable by induction on k, using Lemma 2.1 for the induction step.

We can also easily compute an exact upper bound on the cost of `ReadMax` and `WriteMax` on a $\texttt{MaxReg}_k$ object. For $k = 0$, both `ReadMax` and `WriteMax` perform no operations. For larger k, each `ReadMax` operation performs one register read and then recurses to perform a single `ReadMax` operation on a $\texttt{MaxReg}_{k-1}$ object, while each `WriteMax` performs either a register read or a register write plus at most one recursive call to `WriteMax`. Thus:

THEOREM 2.2. *A* $\texttt{MaxReg}_k$ *object implements a linearizable max register for which every* `ReadMax` *operation requires exactly* k *register reads, and every* `WriteMax` *operation requires at most* k *register operations.*

In terms of the size of the max register, operations on a max register that supports m values, where $2^{k-1} < m \leq 2^k$ values, each take at most $\lceil \log m \rceil$ steps. Note that this cost does not depend on the number of processes n; indeed, it is not hard to see that this implementation works even with infinitely many processes.

2.2. Using an unbalanced binary tree

In order to implement max registers that support unbounded values, we use unbalanced binary trees.

Bentley and Yao [1976] provide several constructions of unbalanced binary trees with the property that the i-th leaf sits at depth $O(\log i)$. The simplest of these, called B_1, constructs the tree by encoding each positive integer using a modified version of a classic variable-length code known as the Elias delta code [Elias 1975]. In this code, each positive integer $N = 2^k + \ell$ with $0 \leq \ell < 2^k$ is represented by the bit sequence $\delta(N) = 1^k 0 \beta(\ell)$, where $\beta(\ell)$ is the k-bit binary expansion of ℓ. The first few such encodings are $0, 100, 101, 11000, 11001, 11010, 11011, 1110000, \ldots$. If we interpret a leading 0 bit as a direction to the left subtree and a leading 1 bit as a direction to the right subtree, this gives a binary tree that consists of an infinitely long rightmost path (corresponding to the increasingly long prefixes 1^k), where the i-th node in this path ($i = 1, 2, \ldots$) has a left subtree that is a balanced binary tree with 2^{i-1} leaves. (A similar construction is used by Attiya and Fouren [2001].)

Let us consider what happens if we build a max register using the B_1 tree (see Figure 2). A `ReadMax` operation that reads the value v will follow the path corresponding to $\delta(v+1)$, and in fact will read precisely this sequence of bits from the switch registers in each recursive max register along the path. This gives a cost to read value v that is

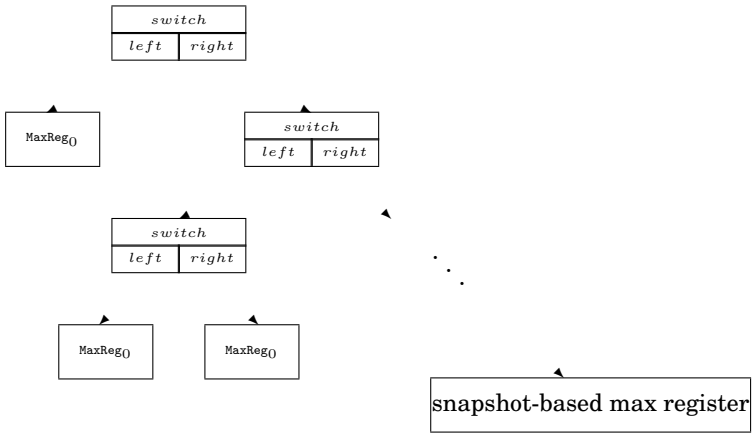

Fig. 2. An unbalanced max register.

equal to $|\delta(v+1)| = 2\lceil\log(v+1)\rceil + 1$. Similarly, the cost of `WriteMax`(v) will be at most $2\lceil\log(v+1)\rceil + 1$.

Both of these costs are unbounded for unbounded values of v. For `ReadMax` operations, there is an additional complication: repeated concurrent `WriteMax` operations might set each switch just before the `ReadMax` reaches it, preventing the `ReadMax` from terminating. Another complication is in proving linearizability, as the induction does not terminate without trickery like truncating the structure just below the last node actually used by any completed operation.

For these reasons, we prefer to backstop the tree with a single snapshot-based max register that replaces the entire subtree at position 1^n, where n is the number of processes. Using this construction, we have:

THEOREM 2.3.
There is a linearizable implementation of `MaxReg` *for which every* `ReadMax` *operation that returns value* v *requires* $\min(2\lceil\log(v+1)\rceil + 1, O(n))$ *register reads, and every* `WriteMax` *operation requires at most* $\min(2\lceil\log(v+1)\rceil + 1, O(n))$ *register operations.*

If constant factors are important, the multiplicative factor of 2 can be reduced to $1 + o(1)$ by using a more sophisticated unbalanced tree; the interested reader should consult [Bentley and Yao 1976] for examples.

Note that the infinite-tree construction does give an obstruction-free algorithm, since any operation does terminate when running alone.

3. COUNTERS

An immediate application of max registers is a construction of a linearizable m-valued counter for n processes with step complexity $O(\min(\log n \log m, n))$ for increment operations and $O(\min(\log m, n))$ for read operations. This is a special case of a general construction of a large class of monotone data structures based on monotone circuits that we present in Section 4, but we describe the construction first in its own terms without depending on the general construction. The complexity bounds are an exponential improvement on the best previously known upper bound of $O(n)$ for exact counting, and on the bound $O(n^{4/5+\epsilon}((1/\delta)\log n)^{O(1/\epsilon)})$, where ϵ is a small constant parameter, for approximate counting which is δ-accurate [Aspnes and Censor 2009].

Pseudocode for the counter is given as Algorithm 2. The counter is structured as a binary tree of max registers. Each max register has an index, which is a bit vector $x_1 x_2 \dots x_k$, where k is the depth of the register in the tree. At the top of the tree is a root max register whose index is the empty sequence $\langle\rangle$. With n processes, the leaves have depth $\lceil \log n \rceil$, and each process is assigned to the leaf whose index $x_1 x_2 \dots x_{\lceil \log n \rceil}$ corresponds to the id of the process, expressed in binary.

Each leaf register records the number of increments done by the corresponding process. Internal nodes of the tree record the total number of increments done by all processes whose leaves lie below the node. These values are, of course, not updated immediately; instead, when a process increments its leaf register, it takes responsibility for propagating the new value up the path to the root. It does so by reading both children of each node along the path, and writing their sum to the node itself, before proceeding to the next node and repeating the process.

A read operation is carried out by simply reading the root node.

```
   shared data: MaxReg objects r[x_1 ... x_k] for each bit vector x_1 ... x_k with
                0 ≤ k ≤ ⌈log n⌉, all initially 0.
 1 procedure CounterIncrement()
 2 begin
 3 |  let x_1 ... x_⌈log n⌉ represent the current process's identity.
 4 |  r[x_1 ... x_⌈log n⌉] ← r[x_1 ... x_⌈log n⌉] + 1
 5 |  for k ← ⌈log n⌉ − 1 down to 0 do
 6 |  |  v_0 ← r[x_1 ... x_k 0]
 7 |  |  v_1 ← r[x_1 ... x_k 1]
 8 |  |  r[x_1 ... x_k] ← v_0 + v_1
 9 |  end
10 end
11 procedure ReadCounter()
12 begin
13 |  return r[⟨⟩]
14 end
```

Algorithm 2: Counter implementation

To prove linearizability, we treat each subtree as a counter object in its own right. We can then consider some counter C rooted at some position $x_1 \dots x_k$ as built from a max register $r[x_1 \dots x_k]$ together with two counters C_0 and C_1, corresponding to the subtrees rooted at $x_1 \dots x_k 0$ and $x_1 \dots x_k 1$, respectively. We can think of the execution of `CounterIncrement` up through the first $\lceil \log n \rceil - k$ iterations of the loop as carrying out an increment operation on one of C_0 or C_1, followed by reading both counters and writing the sum of the values read to $r[x_1 \dots x_k]$. We will show by induction on the height of $r[x_1 \dots x_k]$ in the tree that these internal increment and read operations are linearizable; when we reach the root register $r[\langle\rangle]$, this will show that the full counter is linearizable. The base of this induction is the leaf register $r[x_1 \dots r_{\lceil \log n \rceil}]$, which is trivially linearizable as only one process ever updates this max register, letting us assign as linearization point for increments the corresponding `WriteMax` and for reads the corresponding `ReadMax`.

At higher levels of the tree, we have the following lemma:

LEMMA 3.1. *If C_0 and C_1 are linearizable unit-increment counters, then so is C.*

PROOF. Each increment operation of C is associated with a value equal to $C_0 + C_1$ when it increments C_0 or C_1, treating C_0 and C_1 as atomic counters according to the induction hypothesis.

An increment operation with an associated value v is linearized when a value $v' \geq v$ is first written to the output max register $r[x_1 \dots x_k]$. A read operation is linearized at the point it reads the output max register $r[x_1 \dots x_k]$ (which we consider to be atomic).

To see that the linearization point for increment v occurs within the interval of the operation, observe that no increment can write a value $v' \geq v$ to $r[x_1 \dots x_k]$ before increment v finishes incrementing the relevant sub-counter C_0 or C_1, because before then $C_0 + C_1 < v$. Moreover, the increment v cannot finish before some $v' \geq v$ is first written to $r[x_1 \dots x_k]$, because v itself writes a value $v' \geq v$ before it finishes. Since the read operations are also linearized within their execution interval, this order is consistent with the order of non-overlapping operations.

This gives a valid sequential execution, since we now have exactly one increment operation associated with every integer up to any value read from C, and there are exactly v increment operations ordered before a read operation that returns v. □

THEOREM 3.2. *There is an implementation of a linearizable m-valued unit-increment counter of n processes where a read operation takes $O(\min(\log m, n))$ low-level register operations and an increment operation takes $O(\min(\log n \log m, n))$ low-level register operations.*

PROOF. Linearizability follows from the preceding argument. For the complexity, observe that the read operation has the same cost as `ReadMax`, while an increment operation requires reading and updating $O(1)$ max registers per iteration at a cost of $O(\min(\log m, 2^i))$ for the i-th iteration. The full cost of a write is obtained by summing this quantity as i goes from 0 from $\lceil \log n \rceil$. □

Note that for a polynomial number of increments, an increment takes $O(\log^2 n)$ steps. It is also possible to use unbounded max registers, in which case the value m in the cost of a read or increment is replaced by the current value of the counter.

4. MONOTONE CIRCUITS

In this section, we show how a max register can be used to construct more sophisticated data structures from arbitrary monotone circuits. The cost of operations on the data structure will be a function of the number of gates whose values need to be updated when an input changes and the size of the alphabet.

For each monotone circuit, we can construct a corresponding monotone data structure. This data structure supports operations `WriteInput` and `ReadOutput`, where each `WriteInput` operation updates the value of one of the inputs to the circuit and each `ReadOutput` operation returns the value of one of the outputs. Like the circuit as a whole, the effects of `WriteInput` operations are monotone: attempts to set an input to a value less than or equal to its current value have no effect.

The resulting data structure always provides *monotone consistency*, which is generally weaker than linearizability:

Definition 4.1. A monotone data structure is *monotone consistent* if the following properties hold in any execution:

(1) For each output, there is a total ordering $<$ on all `ReadOutput` operations for it, such that if some operation R_1 finishes before some other operation R_2 starts, then $R_1 < R_2$, and if $R_1 < R_2$, then the value returned by R_1 is less than or equal to the value returned by R_2.

(2) The value v returned by any `ReadOutput` operation satisfies $f(x_1, \ldots, x_k) \leq v$, where each x_i is the largest value written to input i by a `WriteInput` operation that completes before the `ReadOutput` operation starts or the initial value of input i if no such operation exists.
(3) The value v returned by any `ReadOutput` operation satisfies $v \leq f(y_1, \ldots, y_k)$, where each y_i is the largest value written to input i by a `WriteInput` operation that starts before the `ReadOutput` operation completes or the initial value of input i if no such operation exists.

The intuition here is that the values at each output appear to be non-decreasing over time (the first condition), all completed `WriteInput` operations are always observed by `ReadOutput` (the second condition), and no spurious larger values are observed by `ReadOutput` (the third condition). But when operations are concurrent, it may be that some `ReadOutput` operations return intermediate values that are not consistent with any fixed ordering of `WriteInput` operations, violating linearizability (an example is given in Subsection 5.1).

We convert a monotone circuit to a monotone data structure by assigning a max register to each input and each gate output in the circuit. We assume that these max registers are initialized to a default minimum value, so that the initial state of the data structure will be consistent with the circuit. A `WriteInput` operation on this data structure updates an input (using `WriteMax`) and propagates the resulting changes through the circuit as described in Procedure `WriteInput`. A `ReadOutput` operation reads the value of some output node, by performing a `ReadMax` operation on the corresponding output. The cost of a `ReadOutput` operation is the same as that of a `ReadMax` operation: $O(\min(\log m, n))$. The cost of `WriteInput` operation depends on the structure of the circuit: in the worst case, it is $O(Sd \min(\log m, n))$, where S is the number of gates reachable from the input and d is the maximum number of inputs to each gate.

```
1  procedure UpdateGate(g)
2  begin
3  |  Let x_1, ..., x_d be the inputs to g.
4  |  for i ← 1 to d do
5  |  |  y_i ← ReadMax(x_i)
6  |  end
7  |  WriteMax(g, f_g(y_1, ..., y_d))
8  end
9  procedure WriteInput(g, v)
10 begin
11 |  WriteMax(g, v)
12 |  Let g_1, ... g_S be a topological sort of all gates reachable from g.
13 |  for i ← 1 to S do
14 |  |  UpdateGate(g_i)
15 |  end
16 end
17 procedure ReadOutput(g)
18 begin
19 |  return ReadMax(g)
20 end
```

Algorithm 3: Monotone circuit implementation.

THEOREM 4.2. *For any fixed monotone circuit C, the* `WriteInput` *and* `ReadOutput` *operations based on that circuit are monotone consistent.*

PROOF. Consider some execution of a collection of `WriteInput` and `ReadOutput` operations. We think of this execution as consisting of a sequence of atomic `WriteMax` and `ReadMax` operations and use time to refer to points in the execution.

The first clause in Definition 4.1 follows immediately from the linearizability of max registers, since we can just order `ReadOutput` operations by the order of their internal `ReadMax` operations.

For the remaining two clauses, we will jump ahead to the third, upper-bound, clause first. The proof is slightly simpler than the proof for the lower bound, and it allows us to develop tools that we will use for the proof of the second clause.

For each input x_i, let V_i^t be the maximum value written to the register representing x_i at or before time t or its initial value if no such write exists. For any gate g, let $C_g(x_1, \ldots, x_n)$ be the function giving the output of g when the original circuit C is applied to $x_1, \ldots, x_n$ (see Figure 3). For simplicity, we allow C in this case to include internal gates, output gates, and the registers representing inputs (which we can think of as zero-input gates). We thus can define C_g recursively by $C_g(x_1, \ldots, x_n) = x_i$ when $g = x_i$ is an input gate and

$$C_g(x_1, \ldots, x_n) = f_g(C_{g_{i_1}}(x_1, \ldots, x_n), \ldots, C_{g_{i_k}}(x_1, \ldots, x_n))$$

when g is an internal or output gate with inputs $g_{i_1} \ldots g_{i_k}$. Let g^t be the actual output of g in our execution at time t, i.e., the contents of the max register representing the output of g. We claim that for all g and t, $g^t \leq C_g(V_1^t, \ldots, V_n^t)$.

The proof is by induction on t and the structure of C. In the initial state, all max registers are at their default minimum value and the induction hypothesis holds. Suppose now that some max register g changes its value at time t. If this max register represents an input, the new value corresponds to some input supplied directly to `WriteInput`, and we have $g^t = C_g(V_1^t, \ldots, V_n^t)$. If the max register represents an internal or output gate, its value is written during some call to `UpdateGate`, and is equal to $f_g(g_{i_1}^{t_1}, g_{i_2}^{t_2}, \ldots, g_{i_k}^{t_k})$ where each g_{i_j} is some register read by this call to `UpdateGate` and $t_j < t$ is the time at which it is read. Because max register values can only increase over time, we have, for each j, $g_{i_j}^{t_j} \leq g_{i_j}^t = g_{i_j}^{t-1} \leq C_{g_{i_j}}(V_1^{t-1}, \ldots, V_n^{t-1})$ by the induction hypothesis, and the fact that only gate g changes at time t. This last quantity is in turn at most $C_{g_{i_j}}(V_1^t, \ldots, V_n^t)$ as only gate g changes at time t. By monotonicity of f_g we then

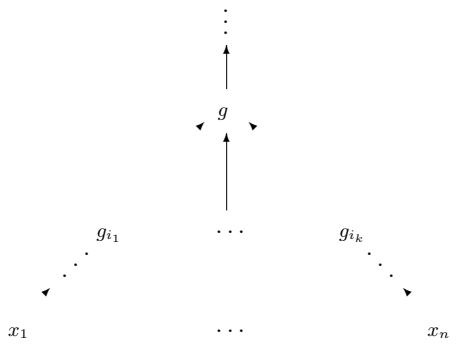

Fig. 3. A gate g in a circuit computes a function of its inputs $f_g(g_{i_1}, \ldots, g_{i_k})$. The inputs to the circuit are $x_1, \ldots, x_n$.

get

$$\begin{aligned} g^t &= f_g(g_{i_1}^{t_1}, g_{i_2}^{t_2}, \ldots, g_{i_k}^{t_k}) \\ &\leq f_g(C_{g_{i_1}}(V_1^t, \ldots, V_n^t), \ldots, C_{g_{i_k}}(V_1^t, \ldots, V_n^t)) \\ &= C_g(V_1^t, \ldots, V_n^t) \end{aligned}$$

as claimed, which completes the proof of clause 3.

We now consider clause 2, which gives a lower bound on output values. For each time t and input x_i, let v_i^t be the maximum value written to the max register representing x_i by a `WriteInput` operation that finishes at or before time t. We wish to show that for any output gate g, $g^t \geq C_g(v_1^t, \ldots, v_n^t)$. As with the upper bound, we proceed by induction on t and the structure of C. But the induction hypothesis is slightly more complicated, in that in order to make the proof go through we must take into account which gate we are working with when choosing which input values to consider.

For each gate g, let $v_i^t(g)$ be the maximum value written to input register x_i by any instance of `WriteInput` that completes `UpdateGate`(g) at or before time t or its initial value if no such operation exists. Our induction hypothesis is that at each time t and for each gate g, $g^t \geq C_g(v_1^t(g), \ldots v_n^t(g))$. Although in general we have $v_i^t \geq v_i^t(g)$, since the set of operations that give v_i^t contains the set of operations that gives $v_i^t(g)$, having $g^t \geq C_g(v_1^t(g), \ldots, v_n^t(g))$ implies $g^t \geq C_g(v_1^t, \ldots, v_n^t)$, as any process that writes to some input x_i that affects the value of g as part of some `WriteInput` operation must complete `UpdateGate`(g) before finishing the operation. The claim holds for $t = 0$, since we assume a consistent initialization of the circuit.

Suppose now that some max register g changes its value at time t or its initial value if no such operation exists. If g is an input, the induction hypothesis holds trivially. Otherwise, consider the set of all `WriteInput` operations that write to g at or before time t. Among these operations, one of them is the last to complete `UpdateGate`(g') for some input g' to g. Let this event occur at time $t' < t$, and call the process that completes this operation p. We now consider the effect of the `UpdateGate`(g) procedure carried out as part of this `WriteInput` operation. Because no other operation completes an `UpdateGate` procedure for any input g_{i_j} to g between t' and t, we have that for each such input and each i, $v_i^t(g_{i_j}) = v_i^{t'}(g_{i_j})$. Since the `ReadMax` operation of each g_{i_j} in p's call to `UpdateGate`(g) occurs after time t', it obtains a value that is at least $g_{i_j}^{t'} \geq C_{g_{i_j}}(v_1^{t'}(g_{i_j}), \ldots, v_n^{t'}(g_{i_j}))$, by the induction hypothesis, and this value is at least $C_{g_{i_j}}(v_1^t(g_{i_j}), \ldots, v_n^t(g_{i_j}))$, by the monotonicity of $C_{g_{i_j}}$ and the previous observation on the relation between $v_i^{t'}(g_{i_j})$ and $v_i^t(g_{i_j})$. But then

$$\begin{aligned} g^t &\geq f_g(C_{g_{i_1}}(v_1^t(g_{i_1}), \ldots, v_n^t(g_{i_1})), \ldots \\ &\qquad C_{g_{i_k}}(v_1^t(g_{i_k}), \ldots, v_n^t(g_{i_k}))) \\ &\geq f_g(C_{g_{i_1}}(v_1^t(g), \ldots, v_n^t(g)), \ldots, C_{g_{i_k}}(v_1^t(g), \ldots, v_n^t(g))) \\ &= C_g(v_1^t(g), \ldots, v_n^t(g)). \end{aligned}$$

□

5. APPLICATIONS

In this section we consider applications of the circuit-based method for building data structures described in Section 4. Most of these applications will be variants on counters, as these are the main example of monotone data structures currently found in the literature. Because we are working over a finite alphabet, all of our counters will be bounded.

5.1. Generalized counters

A *generalized counter* takes an argument to its increment operation, which allows the counter to be incremented by any non-negative amount.

We can construct a generalized counter from a circuit consisting of a binary tree of adders, where each gate in the circuit computes the sum of its inputs and each input to the circuit is assigned to a distinct process to avoid lost updates. This gives an implementation essentially identical to Algorithm 2, except that now an increment by start by adding an arbitrary non-negative value to the leaf register. As in Algorithm 2, the step complexity of an increment is $O(\min(\log n \log m, n))$ and of a read is $O(\min(\log m, n))$.

Unfortunately, unlike in the unit-increment case, it is not hard to see that our generalized counter is not linearizable, even though it satisfies monotone consistency. The reason is that read operations it may return intermediate values that are not consistent with any ordering of the increments.

Here is a small example of a non-linearizable execution, which we present to illustrate the differences between linearizability and monotone consistency. Consider an execution with three writers, and look at what happens at the top gate in the circuit. Imagine that process p_0 executes a `WriteInput` operation with argument 0, p_1 executes a `WriteInput` operation with argument 1, and p_2 executes a `WriteInput` operation with argument 2. Let p_1 and p_2 arrive at the top gate through different intermediate gates g_1 and g_2; we also assume that each process reads g_2 before g_1 when executing `UpdateGate`(g). Now consider an execution in which p_0 arrives at g first, reading 0 from g_2 just before p_2 writes 2 to g_2. Process p_2 then reads g_2 and g_1 and computes the sum 2 but does not write it yet. Process p_1 now writes 1 to g_1 and p_0 reads it, causing p_0 to compute the sum 1 which it writes to the output gate. Process p_2 now finishes by writing 2 to the output gate. If both these values are observed by readers, we have a non-linearizable schedule, as there is no sequential ordering of the increments 0, 1, and 2 that will yield both output values.

However, for restricted applications, we can obtain a fully linearizable object, as shown in the following sections.

5.2. Threshold objects

Another variant of a shared counter that is linearizable is a *threshold object*. This counter allows increment operations, and supports a read operation that returns a binary value indicating whether a predetermined threshold has been crossed. We implement a threshold object with threshold T by having increment operations act as in the generalized counter, and having a read operation return 1 if the value it reads from the output gate is at least T, and 0 otherwise. We show that this implementation is linearizable even with non-uniform increments, where the requirement is that a read operation returns 1 if and only if the sum of the increment operations linearized before it is at least T.

LEMMA 5.1. *The implementation of a threshold object C with threshold T by a monotone data structure with the procedures* `WriteInput` *and* `ReadOutput` *is linearizable.*

PROOF. We use monotone consistency to prove linearizability for the threshold object C. Let C_1 and C_2 be the sub-counters that are added to the final output gate g.

We order read operations according to the ordering implied by monotone consistency, which is consistent with the order of non-overlapping read operations, and implies that once a read operation returns 1 then any following read operation returns 1. We order write operations according to their execution order, which is clearly consistent with the

order of non-overlapping write operations. We then interleave these orders according to the execution order of reads and writes, which implies that this order is consistent with the order of non-overlapping read and write operations.

The interleaving is done while making sure that the sum of increments that are ordered before any read that returns 0 is less than T, and that the sum of increments that are ordered before the first read that returns 1 is at least T. Monotone consistency guarantees that we can do this. For a read operation that returns 0, the value read in g is less than T, therefore the second clause of monotone consistency implies that the sum of all writes that finish before the read starts is less than T. For a read operation that returns 1, the value read in g is at least T, therefore the third clause implies that there enough increment operations that start before this read finishes that have a sum at least T. □

Our proof of Lemma 5.1 does not use the specification of a threshold object, but rather the fact that it is an implementation of a monotone circuit with a binary output. We therefore have:

LEMMA 5.2. *The implementation of any monotone circuit with a binary output by a monotone data structure with the procedures* `WriteInput` *and* `ReadOutput` *is linearizable.*

Note that for any binary-output circuit, we can represent the output using a 1-bit flag initialized to 0 and set to 1 by any `WriteInput` operation that produces 1 as output (essentially, we use the flag as a 2-valued bounded max register). A reader may then do only one operation which accesses that flag and returns its value, i.e., the read operation takes $O(1)$ steps.

5.3. Very cheap snapshots (with expensive updates)

We can use a max register to shift the cost of snapshots to the updating processes, assuming that we execute a bounded number of updates. While this does not reduce the worst-case cost of operations on the snapshot algorithm (updates are still expensive), it may be useful in situations where snapshots are more common than updates and as a counterexample to lower bounds on the cost of bounded snapshots that do not take the cost of updates into account (see also [Israeli et al. 1995]).

Start with some existing snapshot algorithm in which updates take time $S(n)$ and snapshots take time $T(n)$. Augment this algorithm with (a) an array v of m registers, each capable of holding a snapshot, and (b) an m-bounded max register. If the underlying snapshot algorithm does not include a count of the number of updates c_i done by each updating process i in its segment, add this as well.

To perform an update, use the underlying update algorithm and then take a snapshot using the underlying snapshot algorithm. Write this snapshot into $v\left[\sum_{i=1}^{n} c_i\right]$ (note that any two views with the same total count will be identical, so it does not matter if the view is written here more than once). Then write $\sum_{i=1}^{n} c_i$ to the max register. The update fails if $\sum_{i=1}^{n} c_i \geq m$. The cost of an update is $S(n) + T(n) + O(\log m)$. Taking the best known snapshot algorithms [Attiya and Fouren 2001; Inoue and Chen 1994], in which $S(n)$ and $T(n)$ are $O(n)$, the cost of an update is $O(n)$.

To preform a snapshot, read a total count c from the max register and return $v[c]$. The cost of a snapshot is just $O(\log m)$.

Linearizability is easily shown by ordering operations first by the linearization order of their views (equivalently, by total count) and then by the time at which they access the max register.

6. LOWER BOUNDS

Here we give lower bounds on the cost implementing max registers and counters deterministically (Sections 6.1 and 6.2) and using randomization (Section 6.3).

These lower bounds suggest a trade-off between the cost of `WriteMax` and `ReadMax` operations. In Section 6.4 we show that this trade-off can be realized, by giving a probabilistic implementation of a max register that implements `ReadMax` using a single register read at the cost of impractically expensive `WriteMax` operations.

6.1. Lower bound for deterministic max registers

We begin by describing a lower bound of $\min(\lceil \log m \rceil, n-1)$ on the worst-case number of atomic register operations of a `ReadMax` or `ReadCounter` operation in any deterministic asynchronous linearizable implementation of a bounded max register or bounded counter, where m is the number of states of the register or counter and n is the number of processes. For $m \leq 2^{n-1}$, this shows that the balanced tree max-register implementation of Theorem 2.2 has an optimal cost for `ReadMax` operations.

We show that the result holds first for a max register, and observe that the same proof applies to counters. Our proof is based on a covering argument which applies even for *read-once* objects that are only required to be correct in executions with at most one `ReadMax` or `ReadCounter` operation.

THEOREM 6.1. *For any deterministic solo-terminating implementation from atomic registers by n processes of a linearizable max register that supports m values, there is a* `ReadMax` *operation that takes* $\min(\lceil \log m \rceil, n-1)$ *low-level register operations.*

PROOF. To simplify the argument, we consider only a restricted set of executions, and evaluate algorithms based only on their performance (and correctness) on this class of executions. Let S_k be the set of all max register executions consisting of a sequence of (possibly concurrent) executions of the `WriteMax` operation with inputs in the restricted range $\{1, \ldots, k\}$ by processes $\{p_1, \ldots, p_{n-1}\}$, followed by a single `ReadMax` operation by p_0. Let $T(m, n)$ be the worst-case cost of a `ReadMax` operation in any execution in S_m (which also includes all executions in $S_k \subseteq S_m$ for $k \leq m$), and consider some implementation of a max register that is correct on all executions in S_m and that minimizes this cost for given m and n. Then $T(m, n)$, for this implementation, will also give a lower bound on the cost of `ReadMax` operations in arbitrary executions.

Consider the first register read in the `ReadMax` operation of p_0. Because p_0 is deterministic and takes no steps prior to its `ReadMax` operation, any low-level operation it performs depends only on the outcome of its previous low-level operations. Moreover, without loss of generality, we can assume that p_0 performs no register write operations, since there are no steps by other processes after it begins[2]. This implies that the first step by the `ReadMax` of p_0 is a read of a fixed register R, not depending on the `WriteMax` operations preceding it. Let t be the smallest value of k for which there exists an execution in S_k in which some process writes to R. (If no such execution exists, we can omit the read of R from the `ReadMax` operation, contradicting the assumption that the algorithm is optimal.) We use this threshold t to construct two new max register implementations for smaller values of m:

(1) Because t is minimal, if $t > 1$ there is no execution in S_{t-1} in which some process writes to R. It follows that if we restrict the range of values to $\{1, \ldots, t-1\}$, we can omit the read of R from the implementation of `ReadMax` and obtain $T(t-1, n) \leq T(m, n) - 1$, or $T(m, n) \geq T(t-1, n) + 1$.

[2]Recall that we allow the algorithm to be aware of our restricted set of executions, which implies that having write operations during the `ReadMax` of p_0 is not optimal.

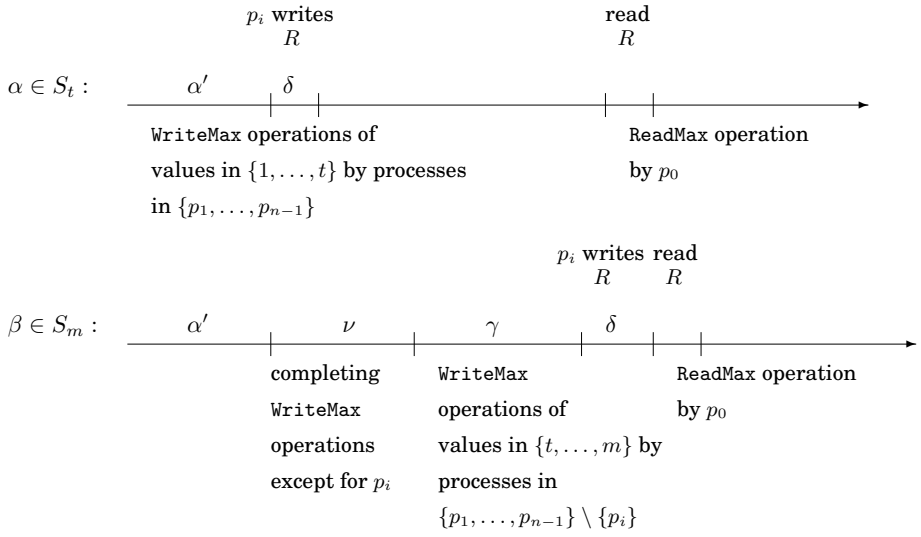

Fig. 4. Proof of Theorem 6.1, Item 2: Although δ covers R in β, the read operation of p_0 returns a value in $\{t, \ldots, m\}$, implying that it distinguishes between $m - t + 1$ values with one less low-level operation.

(2) Additionally, let α be an execution in S_t in which a write to R occurs, and let α' be the prefix of α preceding the first such write and δ the write operation itself (see Figure 4). Let $\alpha'\nu$ be the execution obtained by letting every `WriteMax` operation in progress at the end of α' run to completion, excluding the operation that executes δ. Now consider any execution $\beta = \alpha'\nu\gamma\delta$, where γ is a sequence of non-concurrent `WriteMax` operations with values in the range $\{t, \ldots, m\}$ by processes in $\{p_1, \ldots, p_{n-1}\}$, excluding the process that executes δ. In such executions, a following `ReadMax` operation always observes δ when reading R, but nonetheless returns the largest value written in γ. We can thus obtain an implementation of an $(m-t+1)$-valued $(n-1)$-process max register by initializing the registers to the values they have at the end of $\alpha'\nu$ and replacing the first read in `ReadMax` by a fixed constant. This gives $T(m, n) \geq T(m - t + 1, n - 1) + 1$.

We thus get a recurrence:

$$T(m, n) \geq 1 + \min_{1<t\leq m} \{\max(T(t-1, n), T(m-t+1, n-1))\} ,$$

with $T(1, n) = 0$ and $T(m, 1) = 0$, we show, by double induction on n and m, that the solution to this recurrence is

$$T(m, n) \geq \min(\lceil \log m \rceil, n - 1).$$

For $n = 1$, the bound holds trivially; similarly if $m = 1$. For larger n, and $m > 1$, we have

$$\begin{aligned} T(m, n) &\geq 1 + \min_{1<t\leq m} \{\max(T(t-1, n), T(m-t+1, n-1))\} \\ &\geq 1 + \min_{1<t\leq m} \{\max(T(t-1, n-1), T(m-t+1, n-1))\} \\ &\geq 1 + T(\lceil m/2 \rceil , n-1) \geq 1 + \min(\lceil \log(m/2) \rceil , n-2) \\ &= 1 + \min(\lceil \log m \rceil - 1, n-2) = \min(\lceil \log m \rceil, n-1), \end{aligned}$$

where the third inequality follows from the facts that $T(m,n)$ is a non-decreasing function of both m and n, and that the maximum between $t-1$ and $m-t+1$ is minimized for $t=\lceil m/2 \rceil$. This completes the proof. □

6.2. Lower bound for deterministic counters

In this section we present an alternative lower bound for a k-additive-accurate counter, defined next, using the recurrence technique from Section 6.1 for a bounded max register.

Definition 6.2. A *k-additive-accurate counter* is a counter for which any `ReadCounter` operation returns a value that is within $\pm k$ of the number of `CounterIncrement` operation instances linearized before it.

This lower bound on the cost of implementing bounded max registers deterministically does not extend trivially to counters, for the following reason. It is not known where the `CounterIncrement` operation with the pending low-level operation δ (covering the register R) is linearized in the execution. For max registers this does not matter, since the value written by this operation gets overwritten by larger values, and the reader only has to distinguish between those. However, for a counter, the reader has to return the total number of increments that were linearized before its read operation, therefore must be able to tell whether that increment operation has been linearized before it or not.

We present a lower bound on the cost of implementing counters deterministically, where the key idea is that some additive error is allowed for the `ReadCounter` operation, to account for the accumulating pending operations in he recursion. We describe a lower bound of $\min(n-1, \lceil \log m \rceil - \log(\lceil \log m \rceil + k))$ on the worst-case number of atomic register operations of a `ReadCounter` operation in any deterministic asynchronous linearizable implementation of a bounded counter, where m is the number of states of the counter and n is the number of processes. As in the previous subsection, our proof is based on a covering argument which applies even for *read-once* objects that are only required to be correct in executions with at most one `ReadCounter` operation.

THEOREM 6.3. *For any deterministic solo-terminating implementation from atomic registers by n processes of a linearizable counter that allows m* `CounterIncrement()` *operations, there is a k-accurate* `ReadCounter()` *operation that takes* $\min(n-1, \lceil \log m \rceil - \log(\lceil \log m \rceil + k))$ *low-level register operations.*

PROOF. To simplify the argument, we consider only a restricted set of executions, and evaluate algorithms based only on their performance (and correctness) on this class of executions. Let $S_{n,m,k}$ be the set of all counter executions consisting of a sequence of up to m (possibly concurrent) executions of the `CounterIncrement` operation by processes $\{p_1, \ldots, p_{n-1}\}$, followed by a single k-accurate `ReadCounter()` operation by p_0. Let $T(n,m,k)$ be the worst-case cost of a k-accurate `ReadCounter` operation in any execution in $S_{n,m,k}$ (which also includes all executions in $S_{n,i,k} \subseteq S_{n,m,k}$ for $i \leq m$), and consider some implementation of a counter that is correct on all executions in $S_{n,m,k}$ and that minimizes this cost for a given n,m and k. Then $T(n,m,k)$ will also give a lower bound on the cost of `ReadCounter` operations in arbitrary executions.

Consider the first register read in the `ReadCounter` operation of p_0. Because p_0 is deterministic and takes no steps prior to its `ReadCounter` operation, any low-level operation it performs depends only on the outcome of its previous low-level operations. Moreover, without loss of generality, we can assume that p_0 performs no register write

operations, since there are no steps by other processes after it begins[3]. This implies that the first step by the ReadCounter of p_0 is a read of a fixed register R, not depending on the CounterIncrement operations preceding it.

Let t be the smallest value of m for which there exists an execution in $S_{n,m,k}$ in which some process writes to R. (If no such execution exists, we can omit the read of R from the ReadCounter operation, contradicting the assumption that the algorithm is optimal.) We use this threshold t to construct two new max register implementations for smaller values of m:

(1) Because t is minimal, if $t > 1$ there is no execution in $S_{n,t-1,k}$ in which some process writes to R. It follows that if we restrict the number of increments to $t-1$, we can omit the read of R from the implementation of ReadCounter and obtain $T(n, t-1, k) \leq T(n, m, k) - 1$, or $T(n, m, k) \geq T(n, t-1, k) + 1$.

(2) Additionally, let α be an execution in $S_{n,t,k}$ in which a write to R occurs, and let α' be the prefix of α preceding the first such write and δ the write operation itself. Let $\alpha'\nu$ be the execution obtained by letting every CounterIncrement operation in progress at the end of α' run to completion, excluding the operation that executes δ. Now consider any execution $\beta = \alpha'\nu\gamma\delta$, where γ is a sequence of non-concurrent CounterIncrement operations by processes in $\{p_1, \ldots, p_{n-1}\}$, excluding the process that executes δ. In such executions, a following ReadCounter operation always observes δ when reading R, but nonetheless returns the number of increments finished in $\alpha'\nu\gamma$ up to an error of $k+1$ (since the operation of the process executing δ may not be finished). We can thus obtain an implementation of a $(k+1)$-accurate counter with up to $(m-t+1)$ increments and $(n-1)$-process by initializing the registers to the values they have at the end of $\alpha'\nu$ and replacing the first read in ReadCounter by a fixed constant. This gives $T(n, m, k) \geq T(n-1, m-t+1, k+1)+1$.

We thus get a recurrence:

$$T(n, m, k) \geq 1 + \min_t \left\{\max(T(n, t-1, k), T(n-1, m-t+1, k+1))\right\},$$

with $T(n, m, k) = 0$ for $n = 1$ or $m \leq k$. We show by induction that the solution to this recurrence is

$$T(n, m, k) \geq \min(n-1, \lceil \log m \rceil - \log(\lceil \log m \rceil + k)).$$

For $n = 1$, the bound holds trivially; similarly if $m \leq k$. For larger n, and $m > k$, we have

$$\begin{aligned}
T(n, m, k) &\geq 1 + \min_t \left\{\max(T(n, t-1, k), T(n-1, m-t+1, k+1))\right\} \\
&\geq 1 + T(n-1, \lceil m/2 \rceil, k+1) \geq 1 \\
&\quad + \min(n-2, \lceil \log m/2 \rceil - \log(\lceil \log m/2 \rceil + (k+1))) \\
&= 1 + \min(n-2, \lceil \log m \rceil - 1 - \log(\lceil \log m \rceil - 1 + (k+1))) \\
&= \min(n-1, \lceil \log m \rceil - \log(\lceil \log m \rceil + k)),
\end{aligned}$$

which completes the proof. □

6.3. Lower bound for randomized max registers

We prove a lower bound on randomized implementations of a max register using Yao's Principle [Yao 1977]. The idea is that we can treat any randomized algorithm as a weighted average of deterministic algorithms. A distribution over schedules that gives

[3]Recall that we allow the algorithm to be aware of our restricted set of executions, which implies that having write operations during the ReadCounter of p_0 is not optimal.

a high cost on average for any fixed deterministic algorithm, also gives a high cost on average for any randomized algorithm. Furthermore, if an average schedule gives a high cost for a given randomized algorithm, then there exists some specific schedule that does so.

We state this formally as the following lemma:

LEMMA 6.4. *Suppose there is a probability distribution on inputs x and an oblivious adversary strategy such that $\mathrm{E}[\mathrm{cost}(M(x))] \geq k$ for any deterministic algorithm M. Then for any randomized algorithm M_r (even with global coins) there exists some fixed x such that, with the same oblivious adversary strategy, $\mathrm{E}[\mathrm{cost}(M_r(x))] \geq k$.*

PROOF. Because the adversary is oblivious, its choice of schedule does not depend on the behavior of M_r. So we can simulate M_r by randomly choosing at the start of the execution a deterministic algorithm M_i from the set of all deterministic algorithms $\{M_1, M_2, \dots\}$ obtained by fixing the random bits used by M_r.

We are given that $\mathrm{E}[\mathrm{cost}(M_i(x))|M_i] \geq k$ for any fixed M_i, so $\mathrm{E}[\mathrm{cost}(M_i(x)] \geq k$ when M_i is chosen randomly. It follows that $\mathrm{E}[\mathrm{cost}(M_i(x))|x] \geq k$ for some fixed input x. □

THEOREM 6.5. *For any randomized implementation by n processes of a max register, with probability $1-o(1)$ either some* `WriteMax` *operation takes more than w low-level register operations, some* `ReadMax` *operation takes $\Omega(\log n/\log(w \log n))$ low-level register operations, or some operation fails.*

PROOF. We now show a distribution over schedules that gives a high cost for any deterministic algorithm. We will prove the bound for $m \geq n$. For the general case, we can replace all occurrences of n in the lower bound argument with $\min(m, n)$.

Consider a schedule $\beta_1\beta_2 \dots \beta_{n-1}\rho$, where β_i is a `WriteMax` operation by p_i with value i and ρ is a `ReadMax` operation by p_0. We denote by w the worst-case cost of any `WriteMax` operation, and therefore we allocate w steps to each `WriteMax` operation and have no bound on the number of steps that p_0 takes in ρ.

We modify the above schedule by truncating each `WriteMax` operation randomly. Specifically, we choose a prefix $\beta_i'\delta_i$ of β_i, where δ_i is a single operation that we will delay, with all lengths $\{0, \dots, w-1\}$ for β_i' equally likely. We now construct a family of schedules of the form

$$\begin{aligned}
\alpha_0 &= \rho \\
\alpha_1 &= \beta_1\rho \\
\alpha_2 &= \beta_1'\beta_2\delta_1\rho \\
\alpha_3 &= \beta_1'\beta_2'\beta_3\delta_2\delta_1\rho \\
&\dots
\end{aligned}$$

In each of these schedules α_i, $i \in \{0, \dots, n-1\}$, we run i `WriteMax` operations, where the i-th `WriteMax` operation (if $i > 1$) is run to completion, but previous `WriteMax` operations are used to attempt to cover registers. The covering writes are done in reverse order because it may be that several δ_i operations write to the same location, and we want the earliest value to cover any subsequent values. The key fact is that the location to which δ_i writes and the value it writes to it do not depend on the truncation of the β_j schedules for $j > i$.

We examine the sequence of values read in ρ by p_0 at the end of each α_i, and show that the set of such sequences for successful executions (in which β_i finishes in w steps and ρ returns the correct value i) form a prefix-free code over an alphabet of bounded size. Formally, for every register R we define the number $S_R(i)$ of distinct values that

can appear in each register R at the end of the different schedules α_i. The proof of the following lemma is deferred to the end of this proof:

LEMMA 6.6. *Let R be a register. For every constant c, $S_R(n-1)$ exceeds $1 + cw \log n$ with probability less than $exp(-n^c)$.*

Therefore with high probability, no register exhibits more than $O(w \log n)$ values. For each execution α_i, the reader sees some sequence of values, each of which is chosen from the $O(w \log n)$ possible values for each register. The set of reader executions can be described by a decision tree with $O(w \log n)$-way branching, where each leaf of the tree corresponds to some decision by the reader. For some constant c_0 and sufficiently large n there are at most $\sqrt{n}$ possible leaves in this tree with depth smaller than $c_0 \log n / \log(w \log n)$.

We call an execution α_i *good* if ρ returns i in less than $c_0 \log n / \log(w \log n)$ steps, and otherwise we call α *bad*. The decision tree described above implies that among the n executions α_i, there are at most $\sqrt{n}$ good executions. For the remaining bad executions, i.e., with probability $1 - o(1)$, either some `WriteMax` operation takes more than w steps, some `ReadMax` operation takes $\Omega(\log n / \log(w \log n))$ steps, or some operation fails. This gives the claimed lower bound. □

PROOF OF LEMMA 6.6. The idea is that once R gets covered by δ_j no new values can appear in it in α_i, for $i > j$, i.e., $S_R(i+1) = S_R(i)$. If β_j writes to R then there is a probability of $1/w$ that δ_j indeed covers R, and therefore, we will show that $S_R(n-1)$ is bounded from above by a geometric random variable with parameter $1/w$.

For $i \geq 1$, recall that

$$\alpha_{i-1} = \beta'_1 \dots \beta'_{i-2} \beta_{i-1} \delta_{i-2} \dots \delta_1 \rho$$
$$\alpha_i = \beta'_1 \dots \beta'_{i-2} \beta'_{i-1} \beta_i \delta_{i-1} \delta_{i-2} \dots \delta_1 \rho$$

and consider the following possibilities, in each of which we bound from above the value of $S_R(i)$.

(1) If R is covered by δ_j, for some $j \leq i-2$, then $S_R(i) = S_R(i-1)$.
(2) Otherwise, if R is covered by δ_{i-1} then it could be the case that $S_R(i) = S_R(i-1)+1$ if β_{i-1} has a write to R after δ_{i-1}, which writes some previous value that is replaced in α_i by a new value written by δ_{i-1}. But in this case $S_R(i-1)$ would not have increased compared to $S_R(i-2)$, and therefore the increase in $S_R(i)$ is already accounted for (the inequality in Item 4 for $i-1$ was a strict inequality).
(3) Otherwise, if β_i does not write to R then $S_R(i) = S_R(i-1)$.
(4) Otherwise, $S_R(i) \leq S_R(i-1) + 1$, where the inequality is because it is possible that the value written by β_i is equal to some previous value, and there is a $1/w$ chance that R is covered by δ_i in α_{i+1}.

Therefore, $S_R(n-1)$ is bounded from above by a geometric random variable with parameter $1/w$. The probability that this variable exceeds some value x is less than or equal to

$$(1 - 1/w)^{(x-1)} \leq exp(-1/w)^{(x-1)} = exp(-(x-1)/w).$$

In particular, for every constant c it exceeds $1 + cw \log n$ with probability less than $exp(-n^c)$. □

For $w = \text{polylog}(n)$, the `ReadMax` bound is $\Omega(\log n / \log \log n)$. To get the `ReadMax` bound down to a constant, w must be polynomial in n, and indeed we provide, in the next section, a randomized implementation that achieves this.

6.4. Probabilistic max registers with low read cost

In this subsection we consider a model where the local computation of each process may include an arbitrary number of local coin flips. The coins of different processes are independent, i.e., the processes do not have access to a shared global coin.

It is possible to build a probabilistic version of a max register where a `ReadMax` operation has step complexity 1 but is allowed to return an incorrect value with low probability. This is not intended to be a practical implementation; instead, this is a "counter-example" algorithm demonstrating that the bound on the step complexity of `WriteMax` in the randomized lower bound of Theorem 6.5 is necessary.

The shared data consists of (a) an unbounded `MaxReg` object r, and (b) an array a of $N \gg n$ multi-writer multi-reader atomic registers. Code is given in Algorithm 4.

```
1  procedure ProbabilisticWriteMax(v)
2  begin
3  |  WriteMax(r, v)
4  |  for i ← 1 to N do
5  |  |  a[i] ← ReadMax(r)
6  |  end
7  end
8  procedure ProbabilisticReadMax()
9  begin
10 |  Choose i uniformly at random from {1, ..., N}
11 |  return a[i]
12 end
```

Algorithm 4: Probabilistic max register implementation.

The intuition is that once a `ProbabilisticWriteMax`(v) operation of some process p finishes, all but $n-1$ of the values in a will be at least v. The reason is that p writes a value that is at least v to all N array locations, and each other process can overwrite at most one of these values before re-reading r and obtaining a value at least v thereafter. It follows that `ProbabilisticReadMax` returns a value at least as great as the largest value previously written by a completed `ProbabilisticWriteMax` operation with probability at least $1-(n-1)/N$. It is also not hard to see that it never returns a value that is too large. It follows that `ProbabilisticReadMax` is monotone consistent with probability at least $1-O(n/N)$, which can be made arbitrarily close to 1 at the cost of drastically increasing the step complexity of `ProbabilisticWriteMax`.

THEOREM 6.7. *There are monotone-consistent, probabilistic implementations of* `MaxReg` *in which a* `WriteMax` *operation has step complexity* w*, a* `ReadMax` *operation has step complexity* 1*, and a* `ReadMax` *operation returns an incorrect value with probability* $\min(1, O(n^2/w))$.

PROOF. By applying the preceding analysis where r is a snapshot-based max register and $N = \Theta(w/n)$. (For $w = o(n^2)$, have `WriteMax` and `ReadMax` do nothing.) □

7. DISCUSSION

This paper gives a method for using multi-writer multi-reader registers to construct m-bounded max registers with $\lceil \log m \rceil$ cost per operation, and unbounded max registers with $O(\min(\log v, n))$ cost to read or write the value v. An analog data structure of a *min register* can be implemented in a similar way. We prove a lower bound that shows that the cost of our implementation is optimal.

We note that it is possible to replace the linear snapshot component with the adaptive snapshot implementation of [Attiya et al. 2002] to obtain an unbounded max register construction with $O(\min(\log v, k \log k))$ cost to read or write the value v, where k is the number of processes that participate during this execution.

For randomized implementations we show a lower bound of $\Omega(\log n / \log(w \log n))$ for read operations, where w is the cost of write operations. This leaves open the problem of tightening the randomized lower bound for $m \gg n$, or finding an implementation whose cost depends only on n.

A curious fact is that our randomized lower bound applies even to algorithms supplied with free global coins, since it does not rule out executions in which there are dependencies between the local coins. This puts the lower bound for max register reads higher than the $O(1)$ upper bound on the expected individual step complexity for consensus in a global-coin model.

We use max registers to construct wait-free concurrent data-structures out of any monotone circuit, while satisfying a natural consistency condition we call *monotone consistency*. The cost of a write is $O(Sd \min(\lceil \log m \rceil, O(n)))$, where m is the size of the alphabet for the circuit, S is the number of gates whose value changes as the result of the write, and d is the number of inputs to each gate; the cost of a read is $\min(\lceil \log m \rceil, O(n))$.

As an application, we obtain a simple, linearizable, wait-free counter implementation with a cost of $O(\min(\log n \log v, n))$ to perform an increment and $O(\min(\log v, n))$ to perform a read, where v is the current value of the counter. For polynomially-many increments, these become $O(\log^2 n)$ and $O(\log n)$, respectively, an exponential improvement on the best previously known upper bounds of $O(n)$ for an exact counting and $O(n^{4/5+\epsilon})$ for approximate counting [Aspnes and Censor 2009]. Note that bounding the counters allows us to overcome the linear lower bound of Jayanti et al. [2000], as well as the similar lower bounds by Fich et al. [2005] that hold even with CAS primitives. Whether further improvements are possible is still open.

ACKNOWLEDGMENTS

The authors would like to thank Dana Angluin, David Eisenstat, Danny Hendler, and Hanna Mazzawi for useful discussions, and the anonymous referees for helpful comments and suggestions.

REFERENCES

AFEK, Y., ATTIYA, H., DOLEV, D., GAFNI, E., MERRIT, M., AND SHAVIT, N. 1993. Atomic snapshots of shared memory. *Journal of the ACM 40,* 4 (September), 873–890.

ANDERSON, J. H. 1993. Composite registers. *Distributed Computing 6,* 3, 141–154.

ASPNES, J. AND CENSOR, K. 2009. Approximate shared-memory counting despite a strong adversary. In *SODA '09: Proceedings of the Nineteenth Annual ACM -SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 441–450.

ASPNES, J. AND HERLIHY, M. 1990. Fast randomized consensus using shared memory. *Journal of Algorithms 11,* 3 (Sept.), 441–461.

ATTIYA, H. AND FOUREN, A. 2001. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput. 31,* 2, 642–664.

ATTIYA, H., FOUREN, A., AND GAFNI, E. 2002. An adaptive collect algorithm with applications. *Distributed Computing 15,* 2, 87–96.

ATTIYA, H. AND WELCH, J. 2004. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, Hoboken, New Jersey.

BENTLEY, J. L. AND YAO, A. C.-C. 1976. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett. 5,* 3, 82–87.

ELIAS, P. 1975. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on 21,* 2, 194–203.

FICH, F. E., HENDLER, D., AND SHAVIT, N. 2005. Linear lower bounds on real-world implementations of concurrent objects. In *FOCS '05: Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society, Washington, DC, USA, 165–173.

HERLIHY, M. P. AND WING, J. M. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12,* 3 (July), 463–492.

INOUE, M. AND CHEN, W. 1994. Linear-time snapshot using multi-writer multi-reader registers. In *Distributed Algorithms, 8th International Workshop, WDAG '94, Terschelling, The Netherlands, September 29 - October 1, 1994, Proceedings*. Lecture Notes in Computer Science, vol. 857. Springer, New York, 130–140.

ISRAELI, A., SHAHAM, A., AND SHIRAZI, A. 1995. Linear-time snapshot implementations in unbalanced systems. *Mathematical Systems Theory 28,* 5, 469–486.

JAYANTI, P. 2002. f-arrays: implementation and applications. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*. ACM, New York, NY, USA, 270–279.

JAYANTI, P., TAN, K., AND TOUEG, S. 2000. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing 30,* 2, 438–456.

YAO, A. C.-C. 1977. Probabilistic computations: Toward a unified measure of complexity. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, Los Alamitos, CA, USA, 222–227.

Received .; revised .; accepted .