Max Registers, Counters, and Monotone Circuits

(Preliminary Version)

James Aspnes [*]
Department of Computer Science, Yale University
New Haven, CT
aspnes@cs.yale.edu

Hagit Attiya [†]
Department of Computer Science, Technion
Haifa, Israel
hagit@cs.technion.ac.il

Keren Censor [‡]
Department of Computer Science, Technion
Haifa, Israel
ckeren@cs.technion.ac.il

ABSTRACT

A method is given 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 can be constructed from one-bit multi-writer multi-reader registers at a cost of at most $\lceil \lg m \rceil$ atomic register operations per write or read. The construction takes the form of a binary search tree: applying classic techniques for building unbalanced search trees gives an *unbounded* max register with cost $O(\min(\log v, n))$ to read or write a value v, where n is the number of processes.

It is also shown how a max register can be 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. The cost of a write is bounded by $O(Sd \min(\lceil \lg m \rceil, n))$, 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 $\min(\lceil \lg m \rceil, O(n))$. While the resulting data structure is not linearizable in general, it satisfies a weaker but natural consistency condition. 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, this becomes $O(\log^2 n)$, 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.

Finally, it is shown that the upper bounds are almost optimal. We prove that $\min(\lceil \lg m \rceil, n-1)$ is a lower bound on the worst-case complexity for any solo-terminating deterministic implementation of an m-valued bounded max register, which is exactly equal to the upper bound for $m \leq 2^{n-1}$. The same bound also holds m-valued counters. 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 containing $n-1$ partial write operations and one read operation 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

Keywords

distributed computing, shared memory, max registers, counters, monotone circuits

1. INTRODUCTION

A critical aspect of programming contemporary multiprocessing systems is the implementation of *concurrent data structures*, e.g., getting the maximal 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 [1]. An atomic snapshot object is composed of components, each of which typically updated by a different processes; the components can be atomically scanned. By applying a specific function to the scanned components, 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

[*]Supported in part by NSF grant CNS-0435201.

[†]Supported in part by the *Israel Science Foundation* (grant number 953/06)

[‡]Supported in part by the Adams Fellowship Program of the Israel Academy of Sciences and Humanities.

PODC'09, August 10–12, 2009, Calgary, Alberta, Canada.

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 [7]. Indeed, Jayanti, Tan, Toueg [8] 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. This motivated, e.g., Aspnes and Censor [2] to switch to approximate counting, but even this has $O(n^{4/5+\epsilon})$ cost.

However, careful inspection of the 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.

In this paper, we present polylogarithmic implementations of key data structures with bounded values. We assume the standard model of an asynchronous shared-memory system, where n 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. We measure the cost of an implementation by the number of steps required for an operation.

The cornerstone of our constructions, and our first example, is an implementation of a max register that "beats" the $\Omega(n)$ lower bound of [8] when $\log m$ is $o(n)$. 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.

Instead of simply summing the individual process contributions, as in a snapshot-based implementation of a counter, we can use a tree 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 setting 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 a write is bounded by $O(Sd \min(\lceil \lg m \rceil, n)$, 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 a read is $\min(\lceil \lg m \rceil, O(n))$. 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.

We also show that the $\min(\lceil \lg m \rceil, n-1)$ cost of the max register read operation is the exact worst-case complexity for any solo-terminating deterministic implementation when $m \leq n$. The same bound is also shown to hold for m-valued counters (but in this case it is not known to be tight). 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 containing $n-1$ partial write operations and one read operation in which, with probability $1 - o(1)$, (a) the write operation with maximum value takes more than w steps, (b) the read operation returns an incorrect value, or (c) the read operation takes $\Omega(\log n/(\log w + \log\log n))$ steps. 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.

2. MAX REGISTERS

Our basic data structure is a *max register*, which is an object r that supports two operations. A `WriteMax`(r, t) operation with an argument t that records the value t in r, and a `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 $0..(m-1)$, 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., [7]), 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 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` (Section 2.1). We also describe (in Section 2.2) 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.

2.1 Recursive max registers

We show how to construct a max register recursively from a tree of increasingly large max registers.

The base objects will consist of at most one snapshot-based max register as described in the previous section (used

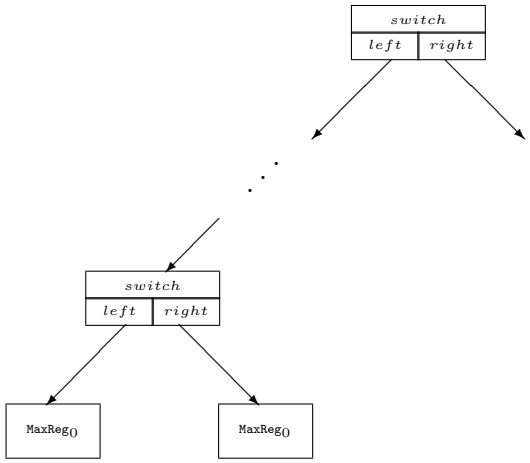

Figure 1: Implementing a max register.

to limit the depth of the tree in the unbounded construction) and a large number of trivial $\texttt{MaxReg}_0$ objects, which are max registers r 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. 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.

Writing a value t to r is by the $\texttt{WriteMax}(r, t)$ procedure, in which the process writes the value t to r.left if $t < m$ and r.switch is off, or otherwise writes the value $t - m$ to r.right and sets the r.switch bit. Reading the maximal value is by the $\texttt{ReadMax}(r)$ procedure, in which the process returns the value it reads from r.left if r.switch is off, and otherwise returns the value it reads from r.right plus m.

```
   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  if t < m then
2      if r.switch = 0 then
3          WriteMax(r.left, t)
4      end
5  end
6  else
7      WriteMax(r.right, t − m)
8      r.switch ← 1
9  end
```

Procedure WriteMax(r,t)

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

LEMMA 1. *If r.left and r.right are linearizable max registers, so is r.*

```
   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  if r.switch = 0 then
2      return ReadMax(r.left)
3  end
4  else
5      return ReadMax(r.right) + m
6  end
```

Procedure ReadMax(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 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}. This preserves the execution order of non-overlapping operations between these two categories, since an operation that starts after an operation in C_{right} finishes cannot be in C_{left}.
2. An operation op in C_{switch} is ordered at the latest time possible before any operation op' that starts after op finishes.
3. Within C_{left} we order the operations according to the time at which they access r.left, i.e., by the order of their respective linearization points.
4. Within C_{right} we order the operations according to the time at which they access r.right (linearization points).

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, 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 again 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 $\texttt{ReadMax}(r.\mathsf{left})$. By the linearizability of $r.\mathsf{left}$, there is a $\texttt{WriteMax}(r.\mathsf{left},t)$ operation op'_w ordered before op' in the linearization of $r.\mathsf{left}$. 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 $r.\mathsf{right}$ 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 $r.\mathsf{switch}$ which is a contradiction to $op \in C_{\mathsf{left}}$. Therefore, $op_w \in C_{\mathsf{left}}$, but this contradicts the linearizability of $r.\mathsf{left}$. 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 $r.\mathsf{right}$. □

Using Lemma 1, we can build a max register whose structure corresponds to an arbitrary binary search 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 discussed next.

2.1.1 Using a balanced binary search tree

To construct a bounded max register of size 2^k, we use a balanced binary search 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 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. *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 $\lceil \lg m \rceil$ 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.

2.1.2 Using an unbalanced binary search tree

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

Bentley and Yao [4] provide several constructions of unbalance binary search 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 [5]. In this code, each positive integer $N = 2^k + \ell$ with $0 \leq \ell < 2^k$ is represented by the bit sequence $\delta(N) = 1^{k-1}0\beta(\ell)$, where $\beta(\ell)$ is the $(k-1)$-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 has a left substree that is a balanced binary search tree with 2^i leaves. (A similar construction is used in [3].)

Let us consider what happens if we build a max register using the B_1 search tree. 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 equal to $|\delta(v+1)| = 2\lceil \lg(v+1) \rceil + 1$. Similarly, the cost of $\texttt{WriteMax}(v)$ will be at most $2\lceil \lg(v+1) \rceil + 1$.

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 bottom 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 3. *There is a linearizable implementation of* `MaxReg` *for which every* `ReadMax` *operation that returns value v requires $\min(2\lceil \lg(v+1) \rceil + 1, O(n))$ register reads, and every* `WriteMax` *operation requires at most $\min(2\lceil \lg(v+1) \rceil + 1, O(n))$ register operations.*

If constant factors are important, the 2 can be reduced to $1 + o(1)$ by using a more sophisticated unbalanced search tree; the interested reader should consult [4] for examples.

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

2.2 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, it demonstrates that the bound on the step complexity of `WriteMax` in the randomized lower bound of Theorem 12 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 the `ProbabilisticWriteMax` and `ProbabilisticReadMax` procedures.

```
1 WriteMax(r, v);
2 for i ← 1 to N do
3     a[i] ← ReadMax(r)
4 end
```

Procedure `ProbabilisticWriteMax`(v)

```
1 Choose i uniformly at random from 1..N;
2 return a[i]
```

Procedure `ProbabilisticReadMax`

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 4. *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 $O(n^2/w)$.*

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.) □

3. MONOTONE CIRCUITS

In this section, we show how a max register can be used to construct more sophisticated data structures from arbitrary monotone circuits.

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. This restriction still allows for an interesting class of data structures, the most useful of which may be the bounded counter described in Section 4.1. The resulting data structure provides **monotone consistency**.

DEFINITION 5. *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.*
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.*

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 Let x_1, ..., x_d be the inputs to g.
2 for i ← 1 to d do
3     y_i ← ReadMax(x_i)
4 end
5 WriteMax(g, f_g(y_1, ..., y_d))
```

Procedure `UpdateGate`(g)

```
1 WriteMax(g, v)
2 Let g_1, ... g_S be a topological sort of all gates reachable
  from g.
3 for i ← 1 to S do
4     UpdateGate(g_i)
5 end
```

Procedure `WriteInput`(g, v)

```
1 return ReadMax(g)
```

Procedure `ReadOutput(g)`

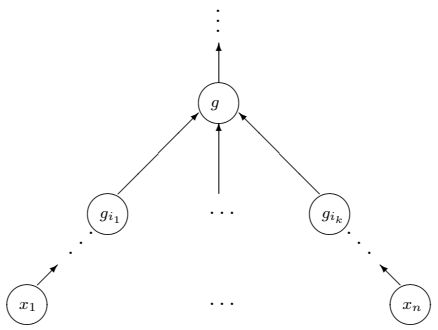

Figure 2: A gate g in a circuit computes a function of its inputs $f_g(g_{i_1}, \ldots, g_{i_k})$. The inputs to the circuit are $x_1, \ldots, x_n$.

THEOREM 6. *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 the total number of such operations completed at any point in the execution.

The first clause in Definition 5 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. 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 2). 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

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

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 get

$$\begin{aligned} g^t &= f_g(g_{i_1}^{t_1}, g_{i_2}^{t_2}, \ldots, g_{i_k}^{t_k}) \\ &\leq f_g(C_{g_1}(V_1^t, \ldots, V_n^t), \ldots, C_{g_k}(V_1^t, \ldots, V_n^t)) \\ &= C_g(V_1^t, \ldots, V_n^t) \end{aligned}$$

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. 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 $v_i^t \geq v_i^t(g)$, 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.

Suppose now that some max register g changes its value at time t. 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 $g_{i_j}^{t'} \geq C_{g_{i_j}}(v_1^{t'}(g_{i_j}), \ldots, v_n^{t'}(g_{i_j})) \geq C_{g_{i_j}}(v_1^t(g_{i_j}), \ldots, v_n^t(g_{i_j}))$, by the induction hypothesis, 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

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

□

4. APPLICATIONS

In this section we consider applications of the circuit-based method for building data structures described in Section 3. 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.

The basic structure we will use is 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. We may consider either bounded or unbounded counters, depending on whether we are using bounded or unbounded max registers. For a bounded counter, we allow only values in the range 0 through $m-1$ for some m; an adder gate whose output would otherwise exceed $m-1$ limits its output to $m-1$. Because the circuit is a tree, a `WriteInput` operation has a particularly simple structure since it need only update gates along a single path to the root; it follows that a `WriteInput` operation costs $O(\min(\log n \log m, n))$ time while a `ReadOutput` operation costs $O(\min(\log m, n))$ time. This is an exponential improvement on the best previously known upper bounds of $O(n)$ for exact counting, and $O(n^{4/5+\epsilon}((1/\delta)\log n)^{O(1/\epsilon)})$, where ϵ is a small constant parameter, for approximate counting which is δ-accurate [2].

If each process is allowed to increase its input by arbitrary values, we get a generalized counter circuit that supports arbitrary non-negative increases to its inputs (the assumption is that each process's input corresponds to the sum of all of its increments so far). Unfortunately, it is not hard to see that the resulting generalized counter is not linearizable, even though it satisfies monotone consistency; the reason is that 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 next subsections.

4.1 Linearizable counters with unit increments

Suppose we consider a standard atomic counter object supporting only read and increment operations, where the increment operation increases the value of the counter by exactly one. This is a special case of the generalized counter discussed above, but here the resulting object is linearizable.

To prove linearizability, we consider the counter C as built of a max register at the root output gate g, which adds up two sub-counters, C_1 and C_2, each supporting half of the processes. Our linearizability proof is then by induction, where the base case is a counter for a single process.

LEMMA 7. *If C_1 and C_2 are linearizable unit-increment counters, then so is C.*

PROOF. Each increment operation of C is associated with a value equal to $C_1 + C_2$ at the time it increments C_1 or C_2, considering that C_1 and C_2 are atomic counters according to the induction hypothesis.

An increment operation with an associated value k is linearized at the first time in which a value $\ell \geq k$ is written to the output max register g. A read operation is linearized at the time it reads the output max register g (which we consider to be atomic).

To see that the linearization point for increment k occurs within the interval of the operation, observe that no increment can write a value $\ell \geq k$ to g before increment k finishes incrementing the relevant sub-counter C_1 or C_2, because before then $C_1 + C_2 < k$. Moreover, the increment k cannot finish before $\ell \geq k$ is first written to g, because k writes a value $\ell \geq k$ 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 clearly 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 k increment operations ordered before a read operation that returns k. □

THEOREM 8. *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 gate at a cost of $O(\min(\log m, 2^i))$ for the i-th gate. The full cost of a write is obtained by summing this quantity as i goes from 0 from $\lceil \lg n \rceil$. □

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.

For general counters (indeed, for any monotone circuit constructed in this way), we only get monotone consistency: the output of a read is at least w and at most W, where w is the number of increments that finish before the read starts, and W is the number of increments that start before the read finishes.

4.2 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 do the same as in the generalized counter, and a read operation returns 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 9. *The implementation of a threshold object C with threshold T by the 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 9 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 10. *The implementation of any monotone circuit with a binary output by the monotone data structure with the procedures* `WriteInput` *and* `ReadOutput` *is linearizable.*

Note that we can obtain a better cost for read operations if we have an additional 1-bit flag instead of the output gate, which is initialized to 0 and set to 1 by any process that increments the counter above the threshold (can be viewed as a 2-valued bounded max register). The reader may then do only one operation which accesses that flag and returns its value.

4.3 Max registers as multiplexers

The linearizability proof in Lemma 1 goes through essentially unchanged even if we replace the objects at the base of the tree with arbitrary linearizable objects. This allows a max register to be used as a multiplexer between different instances of an object, where read operations are directed to the current instance and write operations set the current instance (or fail, if the instance of the write is out of date).

An example of such a multiplexed data structure is the following implementation of a **tagged register** as defined by Subramonian [9]. In a tagged register, each write operation specifies a (tag, value) pair, and the write goes through only if it is the first write with the given tag or larger. Notice that the difference between a tagged register and a max register whose contents is a (tag, value) is that the max register may allow two different values with the same tag to be written. We can implement a tagged register by replacing the leaves in the balanced tree construction of Theorem 2 with CAS objects; here the tree structure handles tags and the CAS objects are used to ensure that at most one value per tag is written.

5. LOWER BOUNDS

5.1 Lower Bound for Deterministic Implementations

We begin by describing a lower bound of $\min(\lceil \lg m \rceil, n-1)$ 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 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 a **read-once** version of the max register that is only required to be correct in executions with at most one `ReadMax` or `ReadCounter` operation.

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 range $1..k$ by processes $p_1..p_{n-1}$, followed by a single `ReadMax` operation by p_n. 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 that is correct on all executions in S_m and that minimizes this cost for a given m and n. Then $T(m, n)$ will also give a lower bound on the cost of `ReadMax` operations in arbitrary executions.

Consider the first register read by p_n. Because p_n 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_n performs no write operations, since there are no steps by other processes after it begins. This implies that the first step by the `ReadMax` of p_n 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 $1..t-1$, 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. 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. 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 $\alpha'\nu\gamma\delta$, where γ is a sequence of non-concurrent `WriteMax` operations with values in the range $t..m$ by processes in $p_1..p_{n-1}$, 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:

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

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

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

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

$$\begin{aligned} T(m,n) &\geq 1 + \min_t \{\max(T(t-1,n), T(m-t+1,n-1))\} \\ &\geq 1 + T(\lceil m/2 \rceil, n-1) \geq 1 + \min(\lceil \lg(m/2) \rceil, n-2) \\ &= 1 + \min(\lceil \lg m \rceil - 1, n-2) = \min(\lceil \lg m \rceil, n-1). \end{aligned}$$

This gives the claimed lower bound:

THEOREM 11. *For any deterministic solo-terminating implementation from atomic registers by n processes of a linearizable max register that supports m values, there is a read operation that takes* $\min(\lceil \lg m \rceil, n-1)$ *low-level register operations.*

Essentially the same argument as in the proof of Theorem 11 gives same lower bound for counters: $T(m,n) \leq \min(\lceil \lg m \rceil, n-1)$ for a counter that allows up to $m-1$ increments. We omit the details for reasons of space.

5.2 Lower Bound for Randomized Implementations

For randomized implementations of max registers, we sketch a lower bound using Yao's Principle [10]. The idea is that we can treat any randomized algorithm as a weighted average of deterministic algorithms. A distribution over schedules that gives 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.

Formally, we consider the set of all deterministic algorithms $M_1, M_2, \ldots$. Given a deterministic oblivious adversary, which supplies a fixed schedule, a randomized algorithm M_r with global coins can be modeled by choosing some fixed M_i randomly at the start of the execution, which happens to deterministically choose the same coin-flip values as M_r does during its execution. If there exists a probability distribution on inputs x such that $E_x[cost(M_i(x))] \geq k$ for all i, then for any randomized algorithm M_r there exists some fixed x such that $cost(M_r(x)) \geq k$.

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 $\beta_1\beta_2\ldots\beta_{n-1}\rho$, where β_i is a write operation by p_i with value i and ρ is a read operation by p_n. We denote by w the worst-case cost of any write operation, and therefore we allocate w steps to each write operation and have no bound on the number of steps that ρ takes.

We modify the above schedule by truncating each write 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 $0..w-1$ for β_i' equally likely. We now construct a family of schedules of the form

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

In each of these schedules, we run $i \in 0..n-1$ write operations, where the i-th write operation (if $i > 1$) is run to completion, but previous write 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 does not depend on the truncation of the β_j schedules for $j > i$.

We examine the sequence of values read by p_n after each α_i, and show that in 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. To do so, we first consider how many distinct values can appear in each register R at the end of the different schedules α_i.

1. β_i doesn't write to R. Then $S(i) = S(i-1)$ and R is still uncovered.

2. β_i writes to R. Then $S(i+1) \geq S(i-1)+1$ (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 now covered by δ_i. If R is covered, then no new values can appear in it, otherwise we continue.

Therefore, $S(\min(m-1,n-1))$ is bounded by a geometric random variable with parameter $1-1/w$. 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, it 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 giving 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 and sufficiently large n there are at most $\sqrt{n}$ possible leaves in this tree with depth smaller than $c \log n / \log(w \log n)$.

If we call an execution α_i *good* if ρ returns i in less than $c \log n / \log(w \log n)$ steps, this implies that among the n executions α_i, there are at most $\sqrt{n}$ good executions. For the remaining bad executions,

1. The reader takes $c \log n / \log(w \log n)$ steps,
2. The write operation in β_i takes more than w steps, or
3. ρ fails to return the correct value.

Therefore with probability $1 - o(1)$, either some write operation takes more than w steps, some read operation takes $\Omega(\log n / \log(w \log n))$ steps, or some operation fails. This gives the claimed lower bound:

THEOREM 12. *For any randomized implementation by n processes of a max register where any write operation takes no more than $O(w)$ low-level register operations, with probability $1 - o(1)$ there exists a read operation which takes $\Omega(\log n / \log(w \log n))$ low-level register operations.*

For $w = \text{polylog}(n)$, the read bound is $\Omega(\log n / \log \log n)$. To get the read bound down to a constant, w must be polynomial in n, and indeed we provide, in Section 2.2, a randomized implementation that achieves this.

6. DISCUSSION

This paper gives a method for using multi-writer multi-reader registers to construct m-bounded max registers with $\lceil \lg m \rceil$ 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. 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 [2]. Note that bounding the counters allows us to overcome the linear lower bound of Jayanti, Tan, and Toueg [8], as well as the similar lower bounds by Fich, Hendler, and Shavit [6] that hold even with CAS primitives. Whether further improvements are possible is still open.

Acknowledgements. The authors would like to thank Dana Angluin, David Eisenstat, and Hanna Mazzawi for useful discussions.

7. REFERENCES

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

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

[3] H. Attiya and A. Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM Journal on Computing*, 31(2):642–664, 2001.

[4] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett.*, 5(3):82–87, 1976.

[5] P. Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, 1975.

[6] F. E. Fich, D. Hendler, and N. Shavit. 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*, pages 165–173, Washington, DC, USA, 2005. IEEE Computer Society.

[7] M. Inoue and W. Chen. Linear-time snapshot using multi-writer multi-reader registers. In *WDAG '94: Proceedings of the 8th International Workshop on Distributed Algorithms*, pages 130–140, London, UK, 1994. Springer-Verlag.

[8] P. Jayanti, K. Tan, and S. Toueg. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing*, 30(2):438–456, 2000.

[9] R. Subramonian. Writing sequential programs for parallel processors: Implementation experience. In *ICCI '92: Proceedings of the Fourth International Conference on Computing and Information*, pages 159–163, Washington, DC, USA, 1992. IEEE Computer Society.

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