Wait-Free Data Structures in the Asynchronous PRAM Model

James Aspnes *
School of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Maurice Herlihy
Digital Equipment Corporation
Cambridge Research Laboratory
One Kendall Square
Cambridge MA, 02139

Abstract

A *wait-free* implementation of a data object in shared memory is one that guarantees that any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes. Much of the literature on wait-free synchronization has focused on the construction of atomic registers, which are memory locations that can be read or written instantaneously by concurrent processes. This model, in which a set of asynchronous processes communicate through shared atomic registers, is sometimes known as asynchronous PRAM. It is known, however, that the asynchronous PRAM model is not sufficiently powerful to construct wait-free implementations of many simple data types such as lists, queues, stacks, test-and-set registers, and others. In this paper, we give an algebraic characterization of a large class of objects that do have wait-free implementations in asynchronous PRAM, as well as a general algorithm for implementing them.

1 Introduction

A *concurrent object* is a data structure shared by asynchronous concurrent processes. An implementation of a concurrent object is *wait-free* if it guarantees that any process will complete any operation in a finite number of steps, regardless of the execution speeds of the other processes. The wait-free condition is a natural property to require of asynchronous systems. It guarantees that no process can be prevented from completing an operation by variations in other processes' speeds, or by undetected halting failures. Even in a failure-free system, a process can encounter unexpected delay by taking a page fault or cache miss, exhausting its scheduling quantum, or being swapped out. Similar problems arise in heterogeneous architectures, where some processors may be inherently faster than others, and some memory locations may be slower to access. The wait-free condition rules out many conventional algorithmic techniques such as busy-waiting, conditional waiting, or critical sections, since the failure or delay of a single process within a critical section will prevent the non-faulty processes from making progress.

The fundamental problem in this area is the following: under what circumstances can we construct a wait-free implementation of one concurrent object from another? Elsewhere [10, 11], we have shown that any object X can be assigned a *consensus number*, which is the largest number of processes (possibly infinite) that can achieve asynchronous consensus [8] by applying operations to a shared X. No object with consensus number n can be implemented by an object with a lower consensus number in a system of n or more processes, but any object with consensus number n is universal (it implements any other object) in a system of n or fewer processes. By computing the consensus numbers of existing synchronization primitives, one can derive an infinite hierarchy of successively more powerful synchronization primitives.

In this paper, we extend our earlier results by investigating the class of objects that have wait-free implementations using only atomic *read* and *write* operations applied to individual memory cells. This model is sometimes known as *asynchronous PRAM* [7, 9]. Many researchers have investigated techniques for constructing such memory cells, called *atomic registers*, from simpler primitives [5, 6, 14, 17, 19, 21, 22, 24]. Despite the impressive amount of intellectual energy that has been applied to these constructions, it is not difficult to show that atomic registers have consensus number 1, and thus the asynchronous PRAM model is too weak to support

*J. Aspnes is funded by an NSF graduate fellowship.

wait-free implementations of any object with a higher consensus number, including common data types such as sets, queues, stacks, priority queues, or lists, most if not all the classical synchronization primitives, such as *test-and-set*, *compare-and-swap*, and *fetch-and-add*, and simple memory-to-memory operations such as *move* or *swap*. These observations raise an intriguing question: what, if anything, are atomic registers good for?

In this paper, we give a new characterization of a wide class of objects that *do* have wait-free implementations in the asynchronous PRAM model. This characterization is algebraic in nature, in the sense that it is expressed in terms of simple commutativity and overwriting properties of the data type's sequential specification. We present a technique for transforming a sequential object implementation into an n-process wait-free implementation requiring a worst-case synchronization overhead of $O(n^2)$ reads and writes per operation. Examples of objects that can be implemented in this way include counters, logical clocks [15], and certain kinds of set abstractions.

2 The Model

Informally, our model of computation consists of a collection of sequential threads of control called *processes* that communicate through shared data structures called *objects*. Each object has a *type*, which defines a set of possible *states* and a set of primitive *operations* that provide the only means to manipulate that object. Each process applies a sequence of operations to objects, issuing an invocation and receiving the associated response. The basic correctness condition for concurrent systems is *linearizability* [12, 13]: although operations of concurrent processes may overlap, each operation appears to take effect instantaneously at some point between its invocation and response. In particular, operations that do not overlap take effect in their "real-time" order.

2.1 I/O Automata

Formally, we model objects and processes using a simplified form of I/O automata [18]. Because the wait-free condition does not require any fairness or liveness conditions, and because we consider only finite sets of processes and objects, we do not make use of the full power of the I/O automata formalism. (For brevity, our algorithms are expressed using pseudocode, although it is straightforward to translate this notation into automata definitions.)

An *I/O automaton A* is a non-deterministic automaton with the following components[1]: *States*(A) is a finite or infinite set of states, including a distinguished set of starting states, *In*(A) is a set of *input events*, *Out*(A) is a set of *output events*, *Int*(A) is a set of *internal events*, *Steps*(A) is a transition relation given by a set of triples (s', e, s), where s and s' are states and e is an event. Such a triple is called a *step*, and it means that an automaton in state s' can undergo a transition to state s, and that transition is associated with the event e. If (s', e, s) is a step, we say that e is *enabled* in s'. I/O automata must satisfy the additional condition that inputs cannot be disabled: for each input event e and each state s', there exist a state s and a step (s', e, s).

An *execution fragment* of an automaton A is a finite sequence $s_0, e_1, s_1, \ldots e_n, s_n$ or infinite sequence $s_0, e_1, s_1, \ldots$ of alternating states and events such that each (s_i, e_{i+1}, s_{i+1}) is a step of A. An *execution* is an execution fragment where s_0 is a starting state. A *history fragment* of an automaton is the subsequence of events occurring in an execution fragment, and a *history* is the subsequence occurring in an execution.

A new I/O automaton can be constructed by *composing* a set of compatible I/O automata. (In this paper we consider only finite compositions.) A set of automata are *compatible* if they share no output or internal events. A state of the composed automaton S is a tuple of component states, and a starting state is a tuple of component starting states. The set of events of S, *Events*(S), is the union of the components' sets of events. The set of output events of S, *Out*(S), is the union of the components' sets of output events; the set of internal events, *Int*(S), is the union of the components' sets of internal events; and the set of input events of S, *In*(S), is *In*(S) − *Out*(S), all the input events of S that are not output events for some component. A triple (s', e, s) is in *Steps*(S) if and only if, for all component automata A, one of the following holds: (1) e is an event of A, and the projection of the step onto A is a step of A, or (2) e is not an event of A, and A's state components are identical in s' and s. Note that composition is associative. If H is a history of a composite automaton and A a component automaton, $H|A$ denotes the subhistory of H consisting of events of A.

2.2 Concurrent Systems

A *concurrent system* is a set of processes and a set of objects. *Processes* represent sequential threads of control, and *objects* represent data structures shared by processes. A process P is an I/O automaton with output events INVOKE(P, op, X), where *op* is an operation [2] of object X, and input events RESPOND(P, res, X), where *res* is a result value. We refer to these events

[1] To remain consistent with the terminology of [13], we use "event" where Lynch and Tuttle use "operation," and "history" where they use "schedule."

[2] *Op* may also include argument values.

as *invocations* and *responses*. Two invocations and responses *match* if their process and object names agree. To capture the notion that a process represents a single thread of control, we say that a process history is *well-formed* if it begins with an invocation and alternates matching invocations and responses. An invocation is *pending* if it is not followed by a matching response. An *object* X has input events INVOKE(P, op, X), where P is a process and op is an operation of the object, and output events RESPOND(P, res, X), where res is a result value. Process and object names are unique, ensuring that process and object automata are compatible.

A *concurrent system* $\{P_1, \ldots, P_n; A_1, \ldots, A_m\}$ is an I/O automaton composed from processes $P_1, \ldots, P_n$ and objects $A_1, \ldots, A_m$, where processes and objects are composed by identifying corresponding INVOKE and RESPOND events. A history of a concurrent system is *well-formed* if each $H|P_i$ is well-formed, and a concurrent system is *well-formed* if each of its histories is well-formed. Henceforth, we restrict our attention to well-formed concurrent systems.

An execution is *sequential* if its first event is an invocation, and it alternates matching invocations and responses. A history is sequential if it is derived from a sequential execution. (Notice that a sequential execution permits process steps to be interleaved, but at the granularity of complete operations.) If we restrict our attention to sequential histories, then the behavior of an object can be specified in a particularly simple way: by giving pre- and postconditions for each operation. We refer to such a specification as a *sequential specification*. In this paper, we consider only objects whose sequential specifications are *total* and *deterministic*: if the object has a pending invocation, then it has a unique matching enabled response. We consider only total operations because it is unclear how to interpret the wait-free condition for partial operations. For example, the most natural way to define the effects of a partial *deq* in a concurrent system is to have it wait until the queue becomes non-empty, a specification that clearly does not admit a wait-free implementation. We consider only deterministic operations because one can always use a deterministic implementation to satisfy a non-deterministic specification, e.g., using the *deq* operation for queues to implement a non-deterministic *choose* operation for sets.

If H is a history, let $complete(H)$ denote the maximal subsequence of H consisting only of invocations and matching responses. Each history H induces a partial "real-time" order $\prec_H$ on its operations: $p \prec_H q$ if the response for p precedes the invocation for q. Operations unrelated by $\prec_H$ are said to be *concurrent*. If H is sequential, $\prec_H$ is a total order. A concurrent system $\{P_1, \ldots, P_n; A_1, \ldots, A_m\}$ is *linearizable* if, each history H can be extended to a well-formed history H', by adding zero or more responses, for each history H, there exists a sequential history S such that:

- For all P_i, $complete(H')|P_i = S|P_i$
- $\prec_H \subset \prec_S$

In other words, the history "appears" sequential to each individual process, and this apparent sequential interleaving respects the real-time precedence ordering of operations. Equivalently, each operation appears to take effect instantaneously at some point between its invocation and its response. A concurrent object A is *linearizable* if, for every history H of every concurrent system $\{P_1, \ldots, P_n; A_1, \ldots, A_j, \ldots, A_m\}$, $H|A_j$ is linearizable. A linearizable object is thus "equivalent" to a sequential object, and its operations can also be specified by simple pre- and postconditions. We restrict our attention to linearizable concurrent systems.

Unlike related correctness conditions such as sequential consistency [16] or strict serializability [20], linearizability is a *local* property: a concurrent system is linearizable if and only if each individual object is linearizable [12, 13]. Henceforth, we restrict our attention to systems and histories that involve a single object.

2.3 Implementations

An *implementation* of an object A is a concurrent system $\{F_1, \ldots, F_n; R\}$, where the F_i are called *front-ends*, and R is called the *representation*. Informally, R is the data structure that implements A, and F_i is the procedure called by process P_i to execute an operation. Each INVOKE(P_i, op, A) is an input event of F_i, and each RESPOND(P_i, res, A) is an output event of F_i. Each input event INVOKE(F_i, op, R) of R is composed with the matching output event of F_i, and each output event RESPOND(F_i, res, R) of R is composed with the matching input event of F_i. An implementation I_j of A_j is *correct* if the two systems are indistinguishable to the ensemble of processes: for every history H of $\{P_1, \ldots, P_n; A_1, \ldots, I_j, \ldots, A_m\}$, there exists a history H' of $\{P_1, \ldots, P_n; A_1, \ldots, A_j, \ldots, A_m\}$, such that $H|\{P_1, \ldots, P_n\} = H'|\{P_1, \ldots, P_n\}$.

In the rest of the paper, we assume the object R is an array of *registers* that provide only *read* and *write* operations. The sequential specification for registers is simple: each *write* takes a value, and each *read* returns the last value written. Values can be of unbounded size.

An implementation is *wait-free* if:

- It has no history in which an invocation of P_i remains pending across an infinite number of steps of F_i.

- If P_i has a pending invocation in a state s, then there exists a history fragment starting from s, consisting entirely of events of F_i and R, that includes the response to that invocation.

The first condition rules out unbounded busy-waiting: a front-end cannot take an infinite number of steps without responding to an invocation. The second condition rules out conditional waiting: F_i cannot block waiting for another process to make a condition true.

2.4 Commuting and Overwriting Invocations

We are now ready to state the algebraic conditions an object must satisfy for us to provide a wait-free implementation. If p is an operation, p_i denotes p's invocation, and p_r its response. We use "$\cdot$" to denote concatenation, and $H \cdot p$ to denote $H \cdot p_i \cdot p_r$.

Definition 1 *Two sequential histories H and H' are* equivalent *if, for all sequential histories G, $H \cdot G$ is legal if and only if $H' \cdot G$ is legal.*

Definition 2 *Invocations p_i and q_i* commute *if, for all sequential histories H, if $H \cdot p$ and $H \cdot q$ are legal then $H \cdot p \cdot q$ and $H \cdot q \cdot p$ are legal and equivalent.*

Definition 3 *Invocation q_i* overwrites *p_i if, for all sequential histories H, if $H \cdot p$ and $H \cdot q$ are legal then $H \cdot p \cdot q$ is legal and equivalent to $H \cdot q$.*

This particular notion of commutativity is due to Weihl [25]. For brevity, we say that two operations commute when their invocations commute.

We will show how to construct a wait-free asynchronous PRAM implementation for any object whose sequential specification satisfies the following property:

Property 1 *For all operations p and q, either p and q commute, or one overwrites the other.*

For example, one data type that satisfies these conditions is the following *counter* data type, providing the following operations:

inc(c: counter, amount: integer)
dec(c: counter, amount: integer)

respectively increment and decrement the counter by a given amount,

reset(c: counter, amount: integer)

reinitializes the counter to *amount*, and

read(c: counter) returns(integer)

returns the current counter value. Note that *inc* and *dec* operations commute, every operation overwrites *read*, and *reset* overwrites every operation. Such a shared counter appears, for example, in randomized shared-memory algorithms [3], and in the implementation of logical clocks [15].

3 Preliminary Lemmas

Lemma 4 *The* overwrites *relation is transitive.*

Proof: Suppose r overwrites q, and q overwrites p.

By the definition of overwrites, there exists a sequential history H such that $H \cdot p$, $H \cdot q$, and $H \cdot r$ are legal, $H \cdot p \cdot q$ is equivalent to $H \cdot q$, and $H \cdot q \cdot r$ is equivalent to $H \cdot r$.

Since operations are total, there exists a response r'_r such that $G = H \cdot p \cdot q \cdot r_i \cdot r'_r$ is legal. Since q overwrites p, G is equivalent to $H \cdot q \cdot r_i \cdot r'_r$. Since $H \cdot q \cdot r$ is legal, and since operations are deterministic, $r_r = r'_r$.

Since r overwrites q, G is equivalent to $H \cdot p \cdot r$. Since q overwrites p, G is also equivalent to $H \cdot r$. We have shown that if $H \cdot p$ and $H \cdot r$ are legal, then $H \cdot p \cdot r$ is legal and equivalent to $H \cdot r$, hence r overwrites p. ∎

Lemma 5 *Let H be a history with operations p, q, r, and s such that p precedes q, r precedes s, and p and s are concurrent. We claim that r must precede q.*

Proof: Since p and s are concurrent, s_i appears before p_r in H. Since r precedes s, r_i and r_r also appear before p_r. Finally, since p precedes q, q_i and q_r appear after p_r, and therefore r and q do not overlap, and r precedes q in H. ∎

For the following definition, processes are ordered by their indices: $P_i < P_j$ if and only if $i < j$.

Definition 6 *An operation p of process P* dominates *operation q of Q if either (1) p overwrites q but not vice-versa, or (2) p and q overwrite each other and $P > Q$.*

The notion of dominance "breaks ties" among mutually overwriting operations.

Lemma 7 *The dominance relation is transitive.*

Proof: Suppose r dominates q, and q dominates p, where operations p, q, and r are respectively executed by processes P, Q, R. By the definition of dominance, r overwrites q, and q overwrites p, hence, by transitivity (Lemma 4), r overwrites p. If p does not overwrite r, we are done, so suppose p also overwrites r. Since p overwrites r and r overwrites q, p overwrites q. Since p and q overwrite one another, and q dominates p, it must be that $P < Q$. Similarly, since q overwrites p, and p overwrites r, q overwrites r, and, by similar reasoning, $Q < R$. It follows that $P < R$, hence r dominates p. ∎

3.1 Precedence and Linearization Graphs

It is convenient to represent a history as a directed acyclic *precedence graph* on completed operations: there is an edge from p to q if and only if p precedes q. The history's *linearization graph* is constructed by augmenting the precedence graph with additional dominance edges. These edges reflect constraints on the ordering of concurrent operations imposed by the algebraic properties of the operations themselves. Given a precedence graph $\mathcal{G}$, the associated linearization graph $L(\mathcal{G})$ is defined by the algorithm shown in Figure 1. Here, $\{p_1, \ldots, p_k\}$ represent the operations sorted in any order consistent with the precedence order. The algorithm constructs a sequence of *intermediate graphs* $\mathcal{L}_{i,j}$, for $0 \leq i < j \leq k$. For brevity, we say that the construction *visits* p_i when it compares p_i to p_j, for $i < j$.

```
$\mathcal{L}_{0,k} := \mathcal{G}$
for $i$ in $1 \ldots k$ do
    $\mathcal{L}_{i,i} := \mathcal{L}_{i-1,k}$
    for $j$ in $i+1 \ldots k$ do
        if $p_i$ dominates $p_j$ and
          there is no path in $\mathcal{L}_{i,j-1}$ from $p_i$ to $p_j$
          then $\mathcal{L}_{i,j} := \mathcal{L}_{i,j-1} \cup p_j \rightarrow p_i$
        elseif $p_j$ dominates $p_i$ and
          there is no path in $\mathcal{L}_{i,j-1}$ from $p_j$ to $p_i$
          then $\mathcal{L}_{i,j} := \mathcal{L}_{i,j-1} \cup p_i \rightarrow p_j$
          else $\mathcal{L}_{i,j} := \mathcal{L}_{i,j-1}$
          end if
        end for
    end for
return $\mathcal{L}_{k,k}$
```

Figure 1: The Linearization Graph Construction

Lemma 8 *If p and q are concurrent in $\mathcal{G}$, and one dominates the other, then there is a path in $L(\mathcal{G})$ from one to the other.*

Proof: When the construction visits the first of p or q, it will add an edge if one does not already exist. ∎

Lemma 9 *If there is no path between p and q in the linearization graph, then they commute.*

Proof: If p and q do not commute, then one dominates the other. ∎

Definition 10 *A* linearization *of a precedence graph $\mathcal{G}$ is a sequential history constructed by a topological sort of $L(\mathcal{G})$.*

Lemma 11 *If $\mathcal{G}$ has a legal linearization, then all linearizations of $\mathcal{G}$ are legal and equivalent.*

Proof: By induction on the number of operations in $\mathcal{G}$. The result is immediate when the graph has a single operation.

Pick an operation p such that p has no outgoing edges in $L(\mathcal{G})$. Let $H = H_1 \cdot p \cdot H_2$ be the legal linearization of $\mathcal{G}$, and $G = G_1 \cdot p \cdot G_2$ any other linearization. Let $\mathcal{G}$' be $\mathcal{G}$ with p removed.

Since p has no outgoing edges in $L(\mathcal{G})$, each operation in H_2 and G_2 is concurrent with p, and hence commutes with p (Lemma 9), so H is equivalent to $H_1 \cdot H_2 \cdot p$. Now, $h' = H_1 \cdot H_2$ is a legal linearization of G', $G' = G_1 \cdot G_2$ is a linearization of G', hence by the induction hypothesis, G' is legal and equivalent to H'. It follows that H is equivalent to $G_1 \cdot G_2 \cdot p$, and since p commutes with each operation in G_2 (see above), H is also equivalent to $G_1 \cdot p \cdot G_2$. ∎

Informally, the purpose of the linearization graph is to ensure that no operation's result is affected by concurrent operations. Linearization graphs owe something to the *serialization graphs* [4] used in database theory, although the technical details are different.

Lemma 12 *Let $\mathcal{G}$ be a precedence graph, and p_0 and p_1 operations concurrent in $\mathcal{G}$, such that there is a path from p_0 to p_1 in the intermediate graph $\mathcal{L}_{i,j}$ in the construction of $L(\mathcal{G})$. Any path of minimal length from p_0 to p_1 in $\mathcal{L}_{i,j}$ contains at most one precedence edge.*

Proof: If there is more then one precedence edge, then there exist operations p, q, r, and s in the path such that p precedes q, there is a path from q to r, and r precedes s. If q precedes s, then the path can be shortened, and therefore p and s are concurrent. By Lemma 5, however, r would then precede q, which contradicts the assumption that there is path from q to r. ∎

Lemma 13 *If p dominates q, and there is a path from p to q in $L(\mathcal{G})$, then there exists an r such that r dominates p and r precedes q.*

Proof: Consider the first intermediate graph in the construction of $L(\mathcal{G})$ to contain a path from p to q. We claim that any path of minimal length from p to q in this graph contains exactly one precedence edge. It cannot contain more than one (Lemma 12), and if it contains none, then q dominates p by transitivity (Lemma 7), which is impossible because p dominates q.

This path traverses operations $p_0 = p, p_1, \ldots, p_m$ and $q_0, q_1, \ldots, q_\ell = q$, such that dominance edges link p_i to p_{i+1} and q_i to q_{i+1}, and p_m precedes q_0. Suppose $p \neq p_k$ and $q \neq q_0$. To construct the paths from p to p_k and q to q_0, the construction must add at least one edge between two of the p_i and at least one edge between two of the q_j. When the construction visits p_i, it adds a dominance edge from p_0 to p_i (unless $p_0 = p_i$), and from p_i to p_m (unless $p_m = p_i$). Although p dominates q, and hence so does p_i, the construction does not add an edge from q to p_i, implying that there must already be a path from p_i to q. Visiting p_i thus completes the path from p to q, implying that p_i must be the last operation visited. A symmetric argument, however, also shows that visiting q_j also completes a path from p to q, implying that q_j must also be the also last operation visited, a contradiction.

Suppose $p_m = p$. Consider the first intermediate graph in the construction of $L(\mathcal{G})$ to contain a path from q_0 to some q', concurrent with q_0, that dominates p. Pick a path of minimal length, and let q'' be the operation immediately before q' in this path. We claim that p and q' must be concurrent, since otherwise the path could be shortened. Lemma 5, however, implies that q'' precedes q_0, contradicting the assumption that there is a path from q_0 to q''.

It follows that $q_0 = q$, and the r in the lemma statement is $p_k \neq p$. ∎

Lemma 14 *Let $\mathcal{G}$ be a precedence graph, p an operation of $\mathcal{G}$ with no outgoing edges, and let $\mathcal{G}'$ be the graph resulting of removing p from $\mathcal{G}$. We claim that $L(\mathcal{G}')$ is a subgraph of $L(\mathcal{G})$.*

Proof: Suppose there is an edge from q to r in $L(\mathcal{G}')$ but not in $L(\mathcal{G})$. Because $\mathcal{G}$ is a subgraph of $\mathcal{G}$, the missing edge must be a dominance edge. The construction for $L(\mathcal{G})$ fails to insert this edge only if it completes a path from r to q before it can add an edge from q to r.

By Lemma 13, there exists r' in $L(\mathcal{G})$ such that r' dominates r, and r' precedes q. Since p does not precede any operations, r' and p are distinct, therefore r' is in G'. Since r' precedes q, the construction visits either r or r' before it visits q. Either way, it constructs a path from r to r' before it compares r and q, thus it completes a path from r to q, a path that does not exist in $L(\mathcal{G}')$. ∎

Lemma 15 *Let p be an operation, and H_1 and H_2 sequential histories such that $H_1 \cdot p$ and $H_1 \cdot H_2$ are legal, and if p dominates any operation q in H_2, then there exists an r in H_2 that precedes q and dominates p. We claim that $H_1 \cdot p \cdot H_2$ is legal.*

Proof: By induction on the length of H_2. The result is immediate if H_2 is empty. Otherwise, H_2 can be written as $q \cdot H_2'$, where q is an operation that p does not dominate. Either q dominates p, in which case the result is immediate, or p and q commute, in which case $H_1 \cdot p \cdot q \cdot H_2'$ is equivalent to $H_1 \cdot q \cdot p \cdot H_2'$, where the latter satisfies the conditions of the lemma, and the result follows from the induction hypothesis. ∎

4 The Algorithm

```
% Shared data
root: array[1..n] of pointer to entry

execute($p_i$: invocation) returns(response)
   % Step 1: construct a response
   view := atomic scan of root array
   H := linearization of view
   e := new entry
   e.invocation := $p_i$
   e.response := $p_r$ such that $H \cdot p_i \cdot p_r$ is legal
   for i in 1 ...n do
      e.preceding[i] := view[i]
      end for
   % Step 2: write out the response
   root[P] := address of e
   return $p_r$
   end execute
```

Figure 2: A Wait-Free Implementation

A wait-free algorithm for implementing an object satisfying Property 1 is shown in Figure 2. The object is represented by its precedence graph. Each operation is represented by an *entry*, a data structure with fields for the invocation, the response, and n pointers to each process's preceding entry. The graph is rooted in an *anchor* array whose P^{th} entry holds a pointer to the entry for process P's most recent operation.

A process executes an operation in two steps:

1. It takes an instantaneous snapshot of the anchor array using the *atomic scan* procedure described in Section 5. It then constructs a linearization graph from the precedence graph rooted at the snapshot array, and then constructs a linearization, called its *view*. Using a sequential implementation of the object, it determines the response to the invocation consistent with the view. It creates an entry for the operation, filling in the response and the precedence edges from the snapshot array.

2. The process updates the precedence graph by storing a pointer to the new entry in its position in the anchor array.

Each of these steps makes a single access to shared data: Step 1 uses the atomic scan algorithm given below, and Step 2 writes a single pointer into the shared *root* array. Informally, this algorithm exploits the commutativity and overwriting properties of operations to ensure that each process sees "enough" of the object state to choose a correct response independently of any operations that may be taking place concurrently. We will show that the shared precedence graph always has a legal linearization.

Lemma 16 *Let $H_1 \cdot p \cdot H_2$ be a linearization of the shared precedence graph $\mathcal{G}$. If p and q are concurrent in $\mathcal{G}$, p dominates q, and q is in H_2, then there exists an r such that r dominates p and r precedes q.*

Proof: Since p and q are concurrent and do not commute, $L(\mathcal{G})$ contains a path from one to the other (Lemma 8). Since p appears before q in the linearization, this path must go from p to q. The result now follows directly from Lemma 13. ∎

An entry that has been initialized but not yet written out is *pending*.

Theorem 17 *The following property is invariant: if the shared precedence graph is linearizable, then it remains linearizable after writing out any pending entry.*

Proof: By induction. The property holds trivially in the object's initial state, when the precedence graph is empty and no entries are pending. The property is preserved when P executes Step 1, since the result of writing out P's entry is linearizable by construction, and the result of writing out any other entry is unchanged.

It remains to check that writing out P's pending entry does not violate linearizability by "invalidating" any other process's pending operation. Suppose P and Q respectively have pending operations p and q. Let $\mathcal{G}$ be the current precedence graph, $\mathcal{G}_p$ the precedence graph after writing out p, $\mathcal{G}_q$ the precedence graph after writing out q, and $\mathcal{G}_{pq}$ the precedence graph after writing out both.

Let $H_1 \cdot p \cdot H_2 \cdot q \cdot H_3$ be a linearization of $L(\mathcal{G}_{pq})$. By Lemma 14, $L(\mathcal{G}_p)$ and $L(\mathcal{G}_q)$ are subgraphs of $L(\mathcal{G}_{pq})$, hence $H_1 \cdot p \cdot H_2 \cdot H_3$ is a linearization of $\mathcal{G}_p$ and $H_1 \cdot H_2 \cdot q \cdot H_3$ a linearization of $\mathcal{G}_q$. By the induction hypothesis, these are both legal sequential histories.

In particular, $H_1 \cdot p$ is legal, $H_1 \cdot H_2 \cdot q \cdot H_3$ is legal, and if p dominates any operation r in $H_2 \cdot q \cdot H_3$, then there exists an r' in $H_2 \cdot q \cdot H_3$ that precedes r and dominates p (Lemma 16). By Lemma 15, $G = H_1 \cdot p \cdot H_2 \cdot q \cdot H_3$ is legal. ∎

Corollary 18 *The object implementation in Figure 2 is linearizable.*

For any particular data type, it is possible to apply type-specific optimizations to discard most of the precedence graph, and to avoid reconstructing the entire linearization graph for each operation. An example of such a construction is given below in Section 6.

```
Scan(P: process, v: value) returns(value)
    scan[P][0] := v ∨ scan[P][0]
    for i in 1...n + 1 do
        for Q in 1...n do
            scan[P][i] := scan[P][i] ∨ scan[Q][i-1]
            end for
        end for
    return scan[P][n+1]
    end Scan
```

Figure 3: The Scan Procedure

5 Atomic Scan

In this section, we show how to take an atomic snapshot scan of an array of multi-reader, single-writer registers in which process P writes the P^{th} array element. It is convenient to cast this problem in slightly more general form: since the array's state does not depend on the order in which distinct processes update their array elements, it is natural to treat the array state as the join in a $\vee$-semilattice of the input values. The snapshot scan simply returns the join of the register values.

Fix a $\vee$-semilattice L; for convenience we will assume that L contains a bottom element $\perp$ such that $\perp \vee x = x$ for all x in L. The atomic scan object has an operation $\text{Write}_L(P, v)$ for each process P and element v of L, and an operation ReadMax(P) for each process P. The serial semantics of the object are straightforward: in any history H, the value returned by a ReadMax(P) operation is the join of the values written by earlier $\text{Write}_L(Q, v)$ operations, for all Q.

The processes share an array $\text{scan}[1 \ldots n][0 \ldots n+1]$ of multi-reader/single-writer atomic registers, where P alone writes to each $\text{scan}[P][i]$. The operations $\text{Write}_L(P, v)$ and ReadMax(P) are each implemented using a stronger primitive operation, Scan(P, v), defined in Figure 3. The Write_L operation is implemented by executing Scan(P, v) and discarding the return value, while the ReadMax operation is implemented by executing Scan($P, \perp$).

5.1 Proof of Correctness

We demonstrate the correctness of the atomic scan algorithm in two steps. First, we show that any two values returned by Scan operations are comparable within the lattice L. Second, we use the lattice ordering of the returned values to order the implemented Write_L and ReadMax operations in any concurrent history H; this ordering will produce an equivalent serial history of the atomic scan object, thus proving linearizability. We use the usual order symbols $<, >, \geq, \leq$ for the semilattice order in L.

An *implementation* history is one in which *high-level*

Scan invocations and responses are interleaved with *low-level read* and *write* invocations and responses in a constrained way: each Scan invocation is separated from its matching response by a sequence of *read* and *write* operations of the same process. Since *read* and *write* operations are linearizable by assumption, we may assume without loss of generality that the subsequence of low-level operations is a sequential history.

Let H be fixed implementation history, p a Scan operation in H executed by process P, and q a Scan operation by P. We use $p[k]$ as an abbreviation for the *write* operation to scan$[P][k]$ executed on behalf of the high-level operation p. We sometimes abuse this notation by using $p[k]$ also to refer to the value it writes. We say that $p[k]$ *directly-sees* $q[k-1]$ if P's *read* of scan$[P][k-1]$ appears after $q[k-1]$ in H. We say that $p[k]$ *sees* $q[l]$ if they lie in the in the reflexive, transitive closure of *directly-sees*. Note that for $p[k]$ to see $q[l]$ it is not enough that $p[k] \geq q[l]$; it must also occur later in time after a sequence of intermediate reads and writes that would allow the value $q[l]$ to be incorporated in the value $p[k]$.

Certain facts about the *sees* relation are important enough to state as lemmas. The proofs are straightforward and are omitted for brevity.

Lemma 19 *If* $i \leq j$*, then* $p[j]$ *sees* $p[i]$.

Lemma 20 *If* $p \prec_H q$ *and* $q[k]$ *and* $p[k]$ *exist, then* $q[k] \geq p[k]$.

It is also not difficult to see that any value written by a process is the join of the values seen by that process; more formally, we state:

Lemma 21 *For any* $p[k]$ *in* H*, if* $0 \leq l < k$*, then* $p[k] = \bigvee \{q[l] \mid p[k] \text{ sees } q[l]\}$.

The following lemma describes the principle on which the atomic scan algorithm depends:

Lemma 22 *If* $p[k]$ *and* $q[k]$ *both appear in* H*, for* $k > 0$*, then either* $p[k]$ *sees* $q[k-1]$ *or* $q[k]$ *sees* $q[k-1]$.

Proof: Suppose $p[k-1]$ precedes $q[k-1]$. Since Q's *read* of scan$[Q][k-1]$ appears after $q[k-1]$, it appears after $p[k-1]$, and $q[k]$ sees $p[k-1]$. otherwise, if $q[k-1]$ precedes $p[k-1]$, then $p[k]$ sees $q[k-1]$. ∎

We now prove the consistency of the atomic scan operation.

Lemma 23 *Either* $p[n+1] \geq q[n+1]$ *or* $q[n+1] \geq p[n+1]$.

Proof: Let p', q' be Scan operations such that $p[n+1]$ sees $p'[0]$, and $q[n+1]$ sees $q'[0]$. We claim that:

$$p[n+1] \geq q'[0] \text{ or } q[n+1] \geq p'[0]. \qquad (1)$$

Let $\{p_0, \ldots, p_{n+1}\}$ be an indexed set of Scan operations (not necessarily distinct) such that $p_0 = p'$, $p_{n+1} = p$, and for each k, $0 < k < n+1$, $p_k[k]$ directly-sees $p_{k-1}[k-1]$. Define $\{q_0, \ldots, q_{n+1}\}$ similarly; the existence of the sets follows from the definition of *sees*.

For each p_k, q_k, where $k > 0$, Lemma 22 implies that either $p_k[k]$ sees $q_k[k-1]$ or $q_k[k]$ sees $p_k[k-1]$, and thus one of p_k or q_k has the property that its $(k-1)^{st}$ *write* is seen by both $p_k[k]$ and $q_k[k]$. Denote this operation by x_k, and the associated process by X_k.

Now consider the indexed set $\{x_0, \ldots, x_{n+1}\}$. By the pigeonhole principle, there exist distinct i and j such that $i < j$ and $X_i = X_j$. If $x_i = x_j$, Lemma 19 immediately implies that $x_j[j-1]$ sees $x_i[i]$.

Otherwise, x_i must precede x_j, because $x_j[j]$ sees either $q_i[i]$ or $p_i[i]$, both of which see $x_i[i-1]$. Thus, by Lemma 20, $x_j[j-1] \geq x_i[j-1]$, but since $j-1 \geq i$ Lemma 19 implies that $x_i[j-1]$ sees $x_i[i]$. Thus in either case $x_j[j-1] \geq x_i[i]$. $p[n+1]$ and $q[n+1]$ see $x_j[j-1]$, and $x_i[i]$ sees one of $p'[0]$, $q'[0]$, showing that Equation 1 holds.

Now suppose that $p[n+1]$ and $q[n+1]$ are incomparable. By Lemma 21, there must then exist a $p'[0]$ which $p[n+1]$ alone sees and a $q'[0]$ which $q[n+1]$ alone sees — contradicting Equation 1. ∎

Theorem 24 *The atomic scan object implementation is linearizable.*

Proof: Consider any two operations x and y. Let $x \prec'_S y$ if either $x[n+1] < y[n+1]$ or $x[n+1] = y[n+1]$, x is a Write$_L$ operation and y is a ReadMax operation. Extend $\prec'_S$ to a total order $\prec_S$; by Lemma 20 $\prec_S$ extends $\prec_H$, and thus we can use it to linearize H. That the resulting sequential history is legal follows directly from Lemma 23. ∎

To implement the atomic snapshot algorithm used in the previous section, we make each value an n-element array of pointers, where the entire array is kept in a single register. (As noted above, numerous techniques exist for constructing large atomic registers from smaller ones.) Each array entry has an associated tag, and the maximum of two entries is the one with the higher tag. The join of two values is the element-wise maximum of the two arrays. The $\perp$ value is just an array whose tags are all zero. P writes the P^{th} position in the *anchor* array by initializing scan$[P][0]$ to an array whose P^{th} element has a higher tag than P's latest entry, and whose other elements have tag zero. (As a simple optimization, the other elements can simply be omitted.)

5.2 Running Time

Each Scan operation requires one *read* and one *write* operation to set scan$[P][0]$, plus n *read* and one *write* operations for each of $n+1$ passes through the loop. Thus a single Scan operation requires a total of n^2+n+1 *read* and $n+2$ *write* operations, where, as usual, n is the number of processes. Some minor gains arise by eliminating superfluous operations that simplify the proof: the very last write (to scan$[P][n+1]$) is unnecessary, as are the reads that a process does of its own registers. After eliminating these operations, a Scan requires n^2-1 *read* and $n+1$ *write* operations.

6 An Example

```
read(c: counter)
    a := atomic scan of c
    result := 0
    for all processes P do
        if P's timestamp is maximal in a
          then result := result + a[P].contrib
          end if
        end for
    return result
    end read

inc(c: counter, amount: integer)
    a := atomic scan of c
    max := entry with maximal timestamp in a
    if my timestamp is maximal
      then a[me].contrib := a[me].contrib + amount
      else a[me].reset_count := max.reset_count
           a[me].reset_sig := max.reset_sig
           a[me].contrib := amount
      end if
    c[me] := a[me]
    end inc

reset(c: counter, amount: integer)
    a := atomic scan of c
    max := entry with maximal timestamp in a
    a[me].contrib := amount
    a[me].reset_count := 1 + max.reset_count
    a[me].reset_sig := me
    c[me] := a[me]
    end reset
```

Figure 4: A Wait-Free Counter Implementation

As an example of how simple optimizations can transform our general algorithm into a more efficient algorithm, we revisit the shared counter example. Here, the precedence graph is represented in extremely compact form. The processes share an n-element array of entries, where each entry has the following fields:

- The *reset count* is an integer, initially zero, used to order reset operations.
- The *reset signature* is the name of the last process observed to reset the counter. It is used to break ties among concurrent resets.
- The *contribution* is an integer representing the sum of the amounts incremented and decremented executed by that process since the last reset.

An entry's *timestamp* is constructed by concatenating the reset count (high-order bits) to the reset signature (low-order bits).

Implementations of the counter operations are shown schematically in Figure 4.

7 Other Related Work

Although the work on atomic registers has a long history, it is only recently that researchers have attempted to formalize the computational power of the resulting model. Cole and Zajicek [7] propose complexity measures and basic algorithms for an "asynchronous PRAM" model in which asynchronous processes communicate through shared atomic registers. Gibbons [9] proposes an asynchronous model in which shared atomic registers are augmented by a form of barrier synchronization. Our work extends these approaches in two ways: we consider data structures rather than the usual numeric or graph algorithms, and we focus on wait-free computation, since we feel that algorithms that require processes to wait for one another are poorly suited to asynchronous models.

We recently learned of two other atomic scan algorithms, developed independently by Afek et al. [1] and by Anderson [2]. The former has time complexity comparable to ours, while the latter uses time exponential in the number of processes. Both of these proposals use bounded counters, while the most straightforward implementation of our scan algorithm uses unbounded counters to represent lattice elements.

An object implementation is *randomized wait-free* if each operation completes in a *fixed* expected number of steps. Elsewhere [3], we have shown that the asynchronous PRAM model is universal for randomized wait-free objects.

8 Remarks

This paper has shown there there is a non-trivial class of objects that have wait-free implementations in the asynchronous PRAM model. This result suggests several open questions. Does every object with consensus number 1 have a wait-free implementation in the asynchronous PRAM model? If so, is there a fixed bound to the number of primitive reads and writes needed to complete an operation, perhaps as a function of n? Or, do there exist objects whose implementations must be finite but unbounded?

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots. Ninth ACM Symposium on Principles of Distributed Computing, to appear., 1990.

[2] Anderson. Composite registers. Technical Report TR-89-25, University of Texas at Austin, September 1989.

[3] J. Aspnes and M.P. Herlihy. Randomized algorithms for wait-free synchronization. Submitted for publication.

[4] P.A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–222, June 1981.

[5] B. Bloom. Constructing two-writer atomic registers. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 249–259, 1987.

[6] J.E. Burns and G.L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 222–231, 1987.

[7] R. Cole and O. Zajiec. The apram: incorporating asynchrony into the pram model. In *Proceedings of the 1989 Symposium on Parallel Algorithms and Architectures*, pages 169–178, Santa Fe, NM, June 1989.

[8] M. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed commit with one faulty process. *Journal of the ACM*, 32(2), April 1985.

[9] P.B. Gibbons. A more practical pram model. In *Proceedings of the 1989 Symposium on Parallel Algorithms and Architectures*, pages 158–168, Santa Fe, NM, June 1989.

[10] M.P. Herlihy. Wait-free synchronization. Accepted for publication, ACM TOPLAS.

[11] M.P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1988.

[12] M.P. Herlihy and J.M. Wing. Linearizabilty: a correctness condition for concurrent objects. Accepted for publication, ACM TOPLAS.

[13] M.P. Herlihy and J.M. Wing. Axioms for concurrent objects. In *14th ACM Symposium on Principles of Programming Languages*, pages 13–26, January 1987.

[14] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[16] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690, September 1979.

[17] L. Lamport. On interprocess communication, parts i and ii. *Distributed Computing*, 1:77–101, 1986.

[18] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, MIT Laboratory for Computer Science, November 1988.

[19] R. Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 232–249, 1987.

[20] C.H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[21] G.L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[22] G.L. Peterson and J.E. Burns. Concurrent reading while writing ii: the multi-writer case. Technical Report GIT-ICS-86/26, Georgia Institute of Technology, December 1986.

[23] A.K. Singh, J.H. Anderson, and M.G. Gouda. The elusive atomic register revisited. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 206–221, August 1987.

[24] P. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proceedings of of the 27th IEEE Symposium on Foundations of Computer Science*, pages 223–243, 1986. See also errata in SIGACT News 18(4), Summer, 1987.

[25] W.E. Weihl. Specification and implementation of atomic data types. Technical Report TR-314, MIT Laboratory for Computer Science, March 1984.