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

May 22, 2000

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.

Contact Author: M. Herlihy, herlihy@crl.dec.com (617) 621-6646.

1 Problem Statement

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, 18, 20, 21, 22, 23]. 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 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 Summary of Results

Details of the formal model [11, 13] are omitted here for brevity. Informally, however, a *concurrent system* consists of a collection of n sequential *processes* that communicate by applying operations to shared typed *objects*. Processes are sequential — each process applies a sequence of operations to objects, alternately issuing an invocation and then receiving the associated response. We make *no* fairness assumptions about processes. A process can halt, or display arbitrary variations in speed. In particular, one process cannot tell whether another has halted or is just running very slowly.

Objects are data structures in shared memory. Each object has a *type*, which defines a set of possible *values* and a set of primitive *operations* that provide the only means to manipulate that object. Each object has a *sequential specification* that defines how the object behaves when its operations are invoked one at a time by a single process. For example, the behavior of a queue object can be specified by requiring that *enq* insert an item in the queue, and that *deq* remove the oldest item present in the queue. In a concurrent system, however, an object's operations can be invoked by concurrent processes, and it is necessary to give a meaning to interleaved operation executions.

An object is *linearizable* [12, 13] if each operation appears to take effect instantaneously at some point between the operation's invocation and response. Linearizability implies that processes appear to be interleaved at the granularity of complete operations, and that the order of non-overlapping operations is preserved. As discussed in more detail elsewhere [13], the notion of linearizability generalizes and unifies a number of *ad-hoc* correctness conditions in the literature, and it is related to (but not identical with) correctness criteria such as sequential consistency [16] and strict serializability [19]. We use linearizability as the basic correctness condition for the concurrent objects constructed in this paper.

An *invocation* consists of an operation name, argument values, and process name, and a *response* consists of a termination condition, result values, and process name. A *history* is a sequence of invocations and responses, subject to simple well-formedness constraints omitted here. An invocation and response *match* if their process names agree. An *operation* in a history is a pair consisting of an invocation and the next matching response. A history is *sequential* if it is a sequence of operations (i.e., matching invocations and responses are not interleaved). It is convenient to treat an object's sequential specification as a prefix-closed set of *legal* sequential histories. In the following, we use "$\cdot$" to denote concatenation of sequences.

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 *Operations p and q* 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 *Operation q* overwrites *p 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 [24]. Both properties are carefully formulated to encompass objects with partial and non-deterministic operations.

In this paper, we 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 might be used, for example, in randomized shared-memory algorithms [3], and for logical clocks [15].

3 The Basic Construction

3.1 Preliminary Definitions

The "real-time" ordering of events induces an irreflexive partial order on operations: p *precedes* q if the response for p precedes the invocation for q. If p and q are unrelated by precedence, they are *concurrent*. It is convenient to think of the precedence order as defining a directed acyclic *precedence graph* on completed operations: there is an edge from p to q if and only if p precedes q.

To reconstruct the object state, we construct a *linearization graph* by augmenting the precedence graph with additional edges. These edges reflect constraints on the ordering of concurrent operations imposed by the algebraic properties of the operations themselves. First, a definition: an operation p of process P *interferes with* operation q of Q if either (1) p overwrites q but not vice-versa, or (2) p overwrites q and $P > Q$.

The linearization graph L associated with a precedence graph G is defined by induction on the number of operations in G.

- The linearization graph of the empty precedence graph is empty.
- Let G be a non-empty precedence graph, let p be an operation of process P having no outgoing edges, and let G' be the precedence graph constructed by removing p and its incoming edges from G. Since G' has fewer operations, it has a well-defined linearization graph L'. The linearization graph L of G is constructing as follows. Construct L'' by adding to L' the precedence edges for p. Let Q be the maximal subgraph of L'' whose vertices consist of all operations q such that there is no path in L'' from q to p. We add the following edges to L''. For each q in Q,
  - If p interferes with q, add an edge from p to q.
  - If q interferes with p and p does not interfere with any operation preceding q in Q, add an edge from p to q.

The linearization graph L is the transitive closure of the result.

Lemma 1 *The linearization graph for G is well-defined; it does not depend on the choice of p.*

Lemma 2 *The linearization graph is acyclic.*

Definition 4 *A* linearization *of L is the sequential history constructed by a topological sort of L.*

Lemma 3 *If L has a legal linearization, then all linearizations of L are legal and equivalent.*

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.

3.2 The Algorithm

The object is represented by a graph whose transitive closure is 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 three steps:

1. The process takes an instantaneous snapshot of the anchor array using the *atomic scan* procedure described in Section 4.

2. The process reconstructs the linearization graph from the precedence graph rooted at the snapshot of the anchor array. It chooses a linearization, called its *view*, and then chooses a response to the invocation consistent with its *view* using a sequential implementation of the object.

3. The process creates an entry for the operation, filling in the response computed in Step 2 and the precedence edges from the anchor array copied in Step 1. It then updates the precedence graph by setting its slot in the *anchor* array to point to the new entry.

In the full paper, we give an inductive proof that any topological sort of the precedence graph's linearization graph is a legal sequential history, hence the object implementation is linearizable. 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.

As described in detail in the full paper, this algorithm can be made considerably more efficient by observing that most of the precedence graph can be discarded, and that it is not necessary to reconstruct the entire linearization graph for each operation. An example of such a construction is given below in Section 5.

4 Atomic Scan

It is convenient to cast the atomic scan problem in a more general form. We can think of a region of memory as representing a pool of information provided by the processes. When the state of the memory does not depend on the order in which values are written, it is natural to treat it as the join in a $\vee$-semilattice of the input values. The atomic scan object simulates a collection of single-writer registers for which it is possible to atomically read 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 $\text{ReadMax}(P,)$ for each process P. The serial semantics of the object are quite straightforward: given any history H the value returned by a $\text{ReadMax}(P,)$ operation in H is equal to the join of all values v such that $\text{Write}_L(P, v)$ appears in H for some processor P.

To implement the atomic scan object, we assume that the processes share between them an array $\text{scan}[1 \ldots n][0 \ldots n+1]$ of multi-reader/single-writer atomic registers, where each register scan$[P][i]$ can be written to by process P. The two operations Write$_L(P, v)$ and ReadMax$(P,)$ are both implemented in terms of a stronger primitive operation Scan(P, v), which is carried out as follows:

1. Read scan$[P][0]$
2. Write $v \vee$ scan$[P][0]$ to scan$[P][0]$.
3. For i from 1 to $n + 1$:
   (a) Read scan$[Q][i-1]$ for all processes Q in arbitrary order.
   (b) Write the $\bigvee_Q$ scan$[Q][i-1]$ to scan$[P][i]$.
4. Return scan$[P][i+1]$

Given the Scan operation, the Write$_L$ operation is implemented by simply ignoring its return value, while the ReadMax operation is just a Scan operation which always writes the value $\perp$. In effect, the Scan operation acts like a Write$_L$ operation followed by a ReadMax operation; we demonstrate this fact formally in the following section.

4.1 Proof of Correctness

We demonstrate the correctness of the atomic scan algorithm in two steps. First, we will show that any two values returned by Scan operations are comparable within the lattice L. Second, we will use the lattice ordering of the set of 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.

Some notation will be useful. The usual order symbols $<, >, \geq, \leq$ will be used for the semilattice order in L. We will assume that we are working from a fixed history H. Since we will be working primarily with the Write events in H, we will abbreviate any event $\langle A, \text{Write}(k), v\rangle$ in H to simply $A[k]$, and will often abuse this notation by using $\wr Ak$ to refer to the value written in addition to the Write event itself. We say that $A[k]$ *directly-sees* $B[k-1]$ if A's Read of scan$[\textit{process}(B)][k-1]$ follows $B[k-1]$ in H. We will say that $A[k]$ *sees* $B[l]$ if $(A[k], B[l])$ is in the reflexive, transitive closure of *directly-sees.* Note that for $A[k]$ to see $B[l]$ it is not enough that $A[k] \geq B[l]$; it must also occur later in time after a sequence of intermediate reads and writes that would allow the value $B[l]$ to be incorporated in the value $A[k]$.

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

Lemma 4 *Let A be an invocation, and let $i \leq j$ be such that $A[i]$ and $A[j]$ both occur. Then $A[j]$ sees $A[i]$.*

Lemma 5 *Let A and B be invocations where $A <_H B$. Let k be such that $A[k]$ and $B[k]$ both exist. Then $\wr Bk \geq \wr Ak$.*

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 6 *Let $A[k]$ occur and let $l < k$, $l \geq 0$. Then $A[k] = \bigvee \{B[l] | A[k] sees B[l]\}$.*

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

Lemma 7 *Let $A[k]$, $B[k]$ both appear in the history for some $k > 0$. Then either $A[k]$ sees $B[k-1]$ or $B[k]$ sees $A[k-1]$.*

Proof: Suppose $A[k-1]$ precedes $B[k-1]$. Then since B's read of scan$[\textit{process}(B)][k-1]$

follows $B[k-1]$ it follows $A[k-1]$ and $B[k]$ sees $A[k-1]$. Alternatively if $B[k-1]$ precedes $A[k-1]$, $A[k]$ sees $B[k-1]$. ∎

We may now prove the consistency of the atomic scan operation.

Lemma 8 *Let A, B be invocations such that $A[n+1]$ and $B[n+1]$ both exist. Then either $A[n+1] \geq B[n+1]$ or $B[n+1] \geq A[n+1]$*

Proof: Let A_0, B_0 be invocations such that $A[n+1]$ sees $A_0[]$ and $B[n+1]$ sees $B_0[]$. We claim that either $A[n+1] \geq B_0[]$ or $B[n+1] \geq A_0[]$. Let $\{A_k\}_{0 \leq k \leq n+1}$ be an indexed set of invocations (not necessarily distinct) such that $A_0 = A_0$, $A_{n+1} = A$, and for each k, $0 < k < n+1$, $A_k[k]$ directly-sees $A_{k-1}[k-1]$. Define $\{B_k\}$ similarly; the existence of the sets follows from the definition of *sees.*

For each A_k, B_k, where $k > 0$, Lemma 7 implies that either $A_k[k]$ sees $B_k[k-1]$ or $B_k[k-1]$, and thus one of A_k or B_k has the property that its $(k-1)$-th write is seen by both $A_k[k]$ and $B_k[k]$. Let X_k stand for this invocation.

Now consider the indexed set $\{X_k\}_{0<k\leq n+1}$. Then there exist distinct i, j such that $X_i = X_j$ or $\mathit{process}(X_i) = \mathit{process}(X_j)$, by the pigeonhole principle.

In the former case $X_i = X_j$, Lemma 4 immediately implies $X_j[j-1]$ sees $X_i[i]$. In the latter case, assume that $i < j$; then that X_i must precede X_j, because $X_j[j]$ sees either $A_i[i]$ or $B_i[i]$, both of which see $X_i[i-1]$. Thus by Lemma 5, $X_j[j-1] \geq X_i[j-1]$, but as $j-1 \geq i$ Lemma 4 gives us $X_i[j-1]$ sees $X_i[i]$. Thus in either case $X_j[j-1] \geq X_i[i]$. But both $A[n+1]$ and $B[n+1]$ see $X_j[j-1]$, and $X_i[i]$ sees one of $A_0[0]$, $B_0[0]$. Thus the claim holds.

Now suppose that $A[n+1]$ and $B[n+1]$ are incomparable; by Lemma 6 there must then exist a $A_0[0]$ which $A[n+1]$ alone sees and a $B_0[0]$ which $B[n+1]$ alone sees— but that would contradict the claim. Thus the lemma holds. ∎

Theorem 5 *The atomic scan object implementation is linearizable.*

Proof: For each invocation A in H, we consider both operations that it may implement, a Write_L operation which we will refer to as A_{Write_L} and a ReadMax operation which we will refer to as A_{ReadMax} (we will delete the extra operation later.) Now consider any two such operations x and y, implemented by invocations X and Y, respectively. Let $x <_S y$ 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. By Lemma 8 $<_S$ can be extended into a total order; furthermore this total order is a superset of $<_H$ by Lemma 5. Thus we can use $<_S$ to linearize H— the actual implemented history is obtained by deleting the extra operations, which have no effect on the object's state. ∎

4.2 Running Time

Each Scan operation requires 1 Read and 1 Write operation to set scan$[P][0]$, plus n Read and 1 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 of these operations can be eliminated; for example, the very last write (to scan$[P][n+1]$) is superfluous, as that register is never read. It does, however, make proving the correctness of the implementation much easier. Depending on the relative cost of storing values locally to a process, it may also make sense to eliminate all reads that a process does of its own registers. If both changes are made, the algorithm require only n^2-1 Read and $n+1$ Write operations.

```
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].contribution
      end
   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].contribution := a[me].contribution + amount
    else a[me].reset_count := max.reset_count
         a[me].reset_signature := max.reset_signature
         a[me].contribution := amount
    end if
  c[me] := a[me]
  end int

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

Figure 1: A Wait-Free Counter Implementation

5 An Example

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 1.

6 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 Attiya 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. We will include a more complete discussion of these algorithms in the full paper, but for now we simply remark that either could be used in our construction.

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.

7 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 interesting open questions. Does every object with consensus number 1 have a wait-free asynchronous PRAM implementation? 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? Do the answers to these questions depend on the number of processes?

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots. Private Communication, 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] 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.

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

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

[21] 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.

[22] 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.

[23] 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.

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