Reasoning with Non-Atomic Memories

*Manhoi Choy** and *Ambuj K. Singh* *

Department of Computer Science
University of California at Santa Barbara
Santa Barbara, CA 93106

August 3, 1993

Abstract

A method for reasoning with non-atomic memory is developed. A program using non-atomic memory is transformed into an equivalent one that uses atomic memory. A number of non-atomic memories including *pipelined RAM*, *causal memory*, and *hybrid consistency* are examined. The approach is illustrated with some examples.

Keywords: concurrency, program correctness, memory consistency conditions.

1 Introduction

The traditional abstraction of shared memory which supported atomic reads and writes has come under increasing scrutiny. Hardware architects seem to agree that atomic memory leads to a large latency that is unacceptable for efficient programming. Based on this observation, non-atomic abstractions of shared memory have been proposed in the literature. These definitions are usually motivated by hardware and their semantics and usefulness from the point of view of a user are far from clear. Coupled with the problems of concurrency and non-determinism, these definitions have the potential of making the programming of concurrent systems very difficult.

This paper examines some existing definitions of non-atomic memories and provides a mechanism for reasoning about them. Instead of designing a new proof system for each kind of non-atomic memory, our approach is to define rules for transforming a program that uses non-atomic memory into an equivalent program that uses atomic memory. The transformed program can then be proved using any of the existing proof systems such as Temporal logic [15] and Unity [6]. Besides providing a technique for reasoning about non-atomic memory, the approach also provides a clear uniform semantics for the non-atomic memories. Traditional approaches toward defining non-atomic memories are based on histories. Systemwide execution histories are considered and those that satisfy the specification are isolated by considering interleavings of events. It may be difficult for users to understand the semantics of non-atomic operations in such an approach. In contrast, the technique proposed here is based on the idea of transforming each non-atomic operation as a set of atomic operations. The motivation is to show that understanding non-atomic memories is, in principle,

*Work supported in part by NSF grant CCR-9008628.

no harder that understanding atomic memories. This approach of transforming a program that uses non-atomic variables into one that uses atomic variables was used earlier by Anderson and Gouda [3] to prove the correctness of programs that use *safe* and *regular* variables [11].

The specific abstractions of non-atomic memory that we examine include *pipelined RAM* [12], *causal memory* [2], TSO and PSO memory models of Sparc [9], and *hybrid consistency* [5]. In each of these cases, suitable auxiliary variables are defined in the process of transformation. These auxiliary variables may be viewed as an abstract implementation of the corresponding kind of memory. The rest of the paper is organized as follows. Sections 2 through 6 examine the different kinds of memory. In each case, rules for transforming each non-atomic read and write are included. In some cases, the transformations are also illustrated with small examples. Section 7 includes a brief discussion.

2 Pipelined RAM

In this kind of non-atomic memory introduced by Lipton and Sandberg [12], every process has its own copy of the shared memory. A read operation is performed by reading this local copy and a write operation is performed by updating the local copy and sending the update to all other processes on FIFO channels. These updates are then executed asynchronously at the remote processes. In order to model this memory, we introduce the following auxiliary variables for each shared variable x and each process p:

- x_p, a local copy of variable x at process p. It is initialized to the initial value of x.
- X_p, a set containing the updates performed by remote processes on variable x. Each tuple in X_p consists of three fields: the updated value, the timestamp of the updating process, and the identity of the updating process. It is initialized to an empty set.
- ts_p, a counter that is used for distinguishing updates by process p. It is initialized to 0.

Each read and write operation of process p is now translated as follows.

1. A read statement $v := x$ is translated to $v := x_p$, i.e., the local copy is read.
2. A write statement $x := m$ is translated to an update of the local copy along with an increment of the local counter, followed by a transmittal of the update to all other processes:

   $x_p, ts_p := m, ts_p + 1;$
   $\langle \forall q : q \neq p : X_q := X_q \cup \{(m, ts_p, p)\} \rangle.$

3. Finally, for each shared variable x, we add a process $M_{p,x}$ that services the remote updates for variable x at process p. Process $M_{p,x}$ examines the contents of set X_p and assigns the values existing there to x_p in a FIFO order:

   repeat
       if $X_p \neq \{\}$ then
           $x_p, X_p := m, X_p - \{(m, ts, q)\}$ where $Min(X_p, (m, ts, q))$
   forever

Predicate $Min(S,t)$ denotes that tuple t is a minimal element in set S in a specified ordering. In this case, tuple t is less than tuple t' provided they mention the same process and the timestamp of t is less than that of t'.

Thus, a given concurrent program that uses *pipelined RAM* is transformed to another equivalent program that uses atomic memory. This new program can be proved using existing logics and proof methods. Though the level of fairness for pipelined RAM has not been discussed explicitly in the literature, for our purposes we assume weak fairness (or *justice*[14]) with respect to each process in the transformed program.

As an example, we consider the program in Figure 1 consisting of two processes and implementing barrier synchronization.

initially $x = y = 0$

p ::	q ::
$x := 1$;	$y := 1$;
$u := y$;	$v := x$;
while $u = 0$ do	while $v = 0$ do
$u := y$	$v := x$
od	od

Figure 1: Program 1

There are two shared variables – x and y, and two local variables – u which is local to process p and v which is local to process q. Process p busy waits until it reads y to be non-zero and process q busy waits until it reads x to be non-zero. Applying our rules for transformation, we get the program in Figure 2.

initially $x_p = y_p = x_q = y_q = ts_p = ts_q = 0 \ \wedge \ Y_p = X_q = \{\}$

p' ::	q' ::
$x_p, ts_p := 1, 1$;	$y_q, ts_q := 1, 1$;
$X_q := \{(1, 1, p)\}$;	$Y_p := \{(1, 1, q)\}$;
$u := y_p$;	$v := x_q$;
while $u = 0$ do	while $v = 0$ do
$u := y_p$	$v := x_q$
od	od

Figure 2: Transformation of Program 1

We also add processes $M_{p,y}$ and $M_{q,x}$ that services the remote updates for variables y and x at processes p and q, respectively. These are shown in Figure 3. The code given below has been simplified by noting that variable x is modified only by process p and variable y is modified only by process q. Furthermore, each process writes to the shared variables at most once.

$M_{p,y}$ ::
  repeat
    if $Y_p = \{(m, ts, q)\}$ then $y_p, Y_p := m, \{\}$
  forever

$M_{q,x}$ ::
  repeat
    if $X_q = \{(m, ts, p)\}$ then $x_q, X_q := m, \{\}$
  forever

Figure 3: Processes $M_{p,y}$ and $M_{q,x}$

For the above system consisting of four processes $p', q', M_{p,y}, M_{q,x}$ it is easily proven that

- $y_p = 1 \Rightarrow y_q = 1$ and $y_q = 1$ leads-to $y_p = 1$, and
- $x_q = 1 \Rightarrow x_p = 1$ and $x_p = 1$ leads-to $x_q = 1$,

which are the properties needed for barrier synchronization.

Pipelined RAM enforces a weak coherence among the values of a shared variable at different processes. If the condition of FIFO message delivery between pairs of processes is dropped, then we get an even weaker kind of non-atomic memory. In the transformation for such a memory, counter ts_p is no longer needed and the transformation rules are simplified as follows. As before, local variable x_p is the local copy of shared variable x at process p. Variable X_p is now a *bag* of updated values for variable x.

1. A read statement $v := x$ is translated to $v := x_p$.
2. A write statement $x := m$ is translated to

   $x_p := m;$
   $\langle \forall q : q \neq p : X_q := X_q \cup \{m\} \rangle.$

3. Process $M_{p,x}$ is as follows:

   repeat
     if $X_p \neq \{\}$ then
       $x_p, X_p := m, X_p - \{m\}$ where $m \in X_p$
   forever

3 Causal Memory

This kind of non-atomic memory was defined by Ahamad, Hutto, and John [2], and later by Ahamad, Burns, Hutto, and Neiger [1]. A causality relation is maintained across read and write events. There are two components of this causality relation: the first is based on program order (this also occurs in the *pipelined RAM* model), and the second is based on the values read by the processes. In order to model this memory, we associate a vector timestamp with each process. Similar to a message passing system, a process affixes its timestamp to each value that it writes and a process updates its timestamp to the timestamp of the value that it reads. Specifically, we use the following two auxiliary variables:

- A vector counter ts_p for each process p that is initialized to all zeros.
- A set X_w for each shared variable x that stores the values written by the processes and a set X_r that stores the values read by the processes. Along with each value, its associated timestamp is also stored. Set X_w is initialized to $\{(x_{init}, \perp)\}$ where x_{init} is the initial value of x and set X_r is initialized to an empty set.

In order to discuss the rules of transformation for this kind of memory, we introduce some terminology. Given any vector timestamp ts, the function nx_p increments the component of ts corresponding to process p. Given any two timestamps ts, ts', the function $max(ts, ts')$ returns a timestamp that has the elementwise maximum of ts and ts'. The rules of translation for process p are as follows.

1. A read statement $v := x$ is translated to the read of any pair (m, ts) in X_w that is *live* [2], i.e, a value which has not been overwritten and for which no later read has returned a different value. Accordingly, let
$$live(m, ts, x, p) \equiv (m, ts) \in X_w \wedge$$
$$\langle \forall m', ts' :: (m', ts') \in X_w \vee ((m', ts') \in X_r \wedge m \neq m') \Rightarrow \neg(ts \prec ts' \prec ts_p) \rangle.$$
In the transformation of the read statement, a live value is read from X_w, the value read along with the current timestamp are inserted into set X_r, and the timestamp of process p is incremented.
$$v, ts_p, X_r := m, nx_p(max(ts, ts_p)), X_r \cup \{(m, max(ts, ts_p))\} \text{ where } live(m, ts, x, p)$$

2. A write statement $x := m$ is translated to an insertion of the update into X_w along with an increment of the local timestamp:
$$X_w, ts_p := X_w \cup \{(m, ts_p)\}, nx_p(ts_p).$$

As an example of the above transformation, consider the program in Figure 4 consisting of three processes p, q, and r and two shared variables x and y. Both the variables are initially 0. Process p updates x to 1. Process q waits until x has been updated and then updates y to 1. Process r waits until y has been updated and then reads x. Variable u is local to process q and variables v and w are local to process r. We would like to show that process r read x only after it has been updated to 1.

The transformed program according to the rules given earlier is shown in Figure 5. It is possible to show that when process q reads x to be 1 and exits from its while loop, $X_w =$

$\{(0,(\perp,\perp,\perp)),(1,(0,0,0))\}$, and X_r contains a value of the kind $(1,(1,m,0))$ for some $m \geq 0$. Subsequently, when process r reads y to be 1 and exits from its while loop, $Y_w = \{(0,(\perp,\perp,\perp)),(1,(1,m+1,0))\}$, Y_r contains a value of the kind $(1,(1,m+1,n))$, and the vector timestamp of r becomes $(1,m+1,n+1)$. Therefore, when process r reads x, the pair $(1,(0,0,0))$ will be the only live value in X_w. Consequently, process r will read the value 1 as desired.

initially $x = y = 0$

p ::	q ::	r ::
$\quad x := 1$	$\quad u := x;$	$\quad v := y$
	$\quad$ while $u = 0$ do	$\quad$ while $v = 0$ do
	$\qquad u := x;$	$\qquad v := y$
	$\quad$ od;	$\quad$ od;
	$\quad y := 1$	$\quad w := x$

Figure 4: Program 2

initially $ts_p = ts_q = ts_r = (0,0,0) \ \wedge\ X_w = Y_w = \{(0,(\perp,\perp,\perp))\} \ \wedge\ X_r = Y_r = \{\}$

p' ::
$\quad X_w, ts_p := X_w \cup \{(1,(0,0,0))\}, nx_p(ts_p)$

q' ::
$\quad u, ts_q, X_r := m, nx_q(max(ts, ts_q)), X_r \cup \{(m, max(ts, ts_q))\}$ where $live(m, ts, x, p)$;
$\quad$ while $u = 0$ do
$\qquad u, ts_q, X_r := m, nx_q(max(ts, ts_q)), X_r \cup \{(m, max(ts, ts_q))\}$ where $live(m, ts, x, p)$
$\quad$ od;
$\quad Y_w, ts_p := Y_w \cup \{(1, ts_q)\}, nx_q(ts_q)$

r' ::
$\quad v, ts_r, Y_r := m, nx_r(max(ts, ts_r)), Y_r \cup \{(m, max(ts, ts_r))\}$ where $live(m, ts, y, r)$;
$\quad$ while $v = 0$ do
$\qquad v, ts_r, Y_r := m, nx_r(max(ts, ts_r)), Y_r \cup \{(m, max(ts, ts_r))\}$ where $live(m, ts, y, r)$
$\quad$ od;
$\quad w, ts_r, X_r := m, nx_r(max(ts, ts_r)), X_r \cup \{(m, max(ts, ts_r))\}$ where $live(m, ts, x, r)$

Figure 5: Transformation of Program 2

4 Sparc Memory Model

The Sparc Memory Model as defined in [9] contains two abstractions of shared memory – TSO (Total Store Ordering) and a weaker variant called PSO (Partial Store Ordering). As in the *Pipelined RAM* model, every process has its own memory. However, the local memory only consists of the values written by the process that have not been copied to the global memory. All updates are eventually performed to the global memory. A read (*load*) attempts to return a value from the local memory, failing which a value is returned from the global memory. A process buffers all its writes (*stores*) to its local memory. In addition to reads and writes, statements that *write-and-read* a value atomically are also allowed. We refer to such a statement as a *swap* statement.

4.1 TSO

For each process p, we maintain a counter ts_p and a local memory L_p. Each tuple in L_p contains three fields – the name of the variable, the associated value, and the timestamp of p when the value was written. The timestamp essentially orders L_p as a FIFO queue. We also maintain a main memory copy $M.x$ for each variable x. The transformation rules for a process p are as follows.

1. A write statement $x := m$ is translated to an insertion of the update in L_p along with an increment of the local timestamp:

   $L_p, ts_p := L_p \cup \{(x, m, ts_p)\}, ts_p + 1.$

2. A read statement $v := x$ is translated to either a read of the most recent value from L_p or a read from the main memory. $L_p|_x$ denotes the set of updates on variable x in set L_p.

   if $L_p|_x \neq \{\}$
   then $v := m$ where $Max^1(L_p|_x, (x, m, ts))$
   else $v := M.x$.

3. A swap statement $swap(v, x)$ is translated into waiting until the local memory is emptied (in order to ensure a FIFO ordering) and then a swap with the main memory:

   wait until $L_p = \{\}$;
   $v, M.x := M.x, v$.

4. Finally, we add a process M_p that copies the contents of the local memory of process p to the main memory in a FIFO manner.

   repeat
       if $L_p \neq \{\}$ then
           $M.x, L_p := m, L_p - \{(x, m, ts)\}$ where $Min(L_p, (x, m, ts))$
   forever

As an example, we consider the implementation of a spin lock using *swap* instructions shown in Figure 6. The *Lock* procedure is executed in order to acquire a lock and the *Unlock* procedure is executed in order to release a lock. These procedures use a shared variable *lock* which is initialized to

[1] Here tuple t is less than tuple t' provided the timestamp of t is less than the timestamp of t'.

0. reg_p is a local register at process p. In order to acquire a lock, a process reads $lock$ until it is 1 and then executes a swap instruction. If successful, the process enters the critical section. Otherwise, the sequence of steps is repeated. (This is the so called test-and-test-and-set implementation of spin locks [4].) To release a lock, a process simply sets $lock$ to 1.

```
initially lock = 1.

  Lock:
      repeat
          repeat
              reg_p := lock
          until reg_p = 1;
          reg_p := 0;
          swap(lock, reg_p)
      until reg_p = 1;

  Critical Section:
      ...
  Unlock:
      lock := 1;
```

Figure 6: Algorithm for process p

Applying our rules of transformation, we obtain the program in Figure 7. In order to prove mutual exclusion, we associate an auxiliary variable cs_p with each process. This variable is initially 0 and is set to the value of reg_p after process p executes the swap statement

$$reg_p, M.lock := M.lock, reg_p \ .$$

Thus, cs_p is set to 1 only after a successful swap statement. When process p resets the lock, the variable is reset to 0. It is possible to prove the following invariant for the program:

$$\Sigma cs_p \ + \ M.lock \ = \ 1.$$

Since cs_p is 1 if process p is in the critical section, the property of mutual exclusion follows. Also note that process M_p copies the updates in L_p in a FIFO manner. Therefore, all updates by process p during the execution of a critical section are copied to the main memory before the next process finds $lock$ to be 1 and enters the critical section.

4.2 *PSO*

This is a weaker abstraction then *TSO* as the local memories are not necessarily FIFO. The programmer is required to issue explicit ordering commands in the form of *stbar* instructions when an ordering is desired. Thus, the local memory is separated into a sequence of blocks of values separated through *stbar* instructions. As in the *TSO* model, a timestamp ts_p is used to separate updates belonging to different blocks. An additional counter c_p is used to order updates belonging initially $M.lock = 1 \ \wedge \ \langle \forall p :: L_p = \{\} \rangle$.

$Lock$:

```
repeat
    repeat
        if L_p|lock ≠ {}
        then reg_p := m where Min(L_p|lock, (lock, m, ts))
        else reg_p := M.lock
    until reg_p = 1;
    reg_p := 0;
    wait until L_p = {};
    reg_p, M.lock := M.lock, reg_p
until reg_p = 1;
```

$Critical\ Section$:

...

$Unlock$:

$L_p, ts_p := L_p \cup \{(lock, 1, ts_p)\}, ts_p + 1;$

M_p ::

```
repeat
    if L_p ≠ {} then
        M.x, L_p := m, L_p − {(x, m, ts)}  where Min(L_p, (x, m, ts))
forever
```

Figure 7: Transformation of Program in Figure 6

to the same block. Each update in the local memory contains an additional field corresponding to this counter. The rules for transformation for a process p are similar to the TSO case.

1. An *stbar* statement is translated to an increment of the local timestamp along with a resetting of the local counter:

   $$ts_p, c_p := ts_p + 1, 0.$$

2. A write statement $x := m$ is translated to an insertion of the update in L_p and an increment of c_p:

   $$L_p, c_p := L_p \cup \{(x, m, ts_p, c_p)\}, c_p + 1.$$

3. A read statement $v := x$ is translated to either a read of the most recent value from L_p or a read from the main memory:

if $L_p|_x \neq \{\}$
then $v := m$ where $Max^2(L_p|_x, (x, m, ts, c))$
else $v := M.x$.

4. A swap statement $swap(x, v)$ is translated into waiting until the local memory does not contain any updates from previous blocks and then a swap with the main memory:

   wait until $\langle \forall x', m', ts', c' : (x', m', ts', c') \in L_p : ts' = ts_p \rangle$;
   $v, M.x := M.x, v$.

5. Finally, we add a process M_p that copies the contents of the local memory of process p to the main memory while preserving the following order: updates belonging to different blocks are copied to the main memory in the order of their timestamps, updates belonging to the same block and on the same variable are copied in the order of their counter values.

   repeat
   if $L_p \neq \{\}$ then
   $M.x, L_p := m, L_p - \{(x, m, ts, c)\}$ where $(x, m, ts, c) \in L_p \wedge$
   $\langle \forall x', m', ts', c' : (x', m', ts', c') \in L_p : ts < ts' \vee (ts = ts' \wedge (x \neq x' \vee c < c'))\rangle$;
   forever

5 Weak Memory

In this kind of non-atomic memory, there is no relationship between different values written to memory; a read operation can return any value written previously. There is no notion of overwriting of values. By itself, this kind of memory is too weak to be of any practical value. However, it can be used in combination with other kinds of memory to define useful consistency conditions. In order to define the semantics of this memory, we associate a set X with each variable x. A write operation inserts a value in the set and a read operation reads any value from the set. Set X is initialized to the initial value of x. For completeness, the translation rules are as follows.

1. A read statement $v := x$ is translated to

   $v := m$ where $m \in X$.

2. A write statement $x := m$ is translated to

   $X := X \cup \{m\}$.

6 Hybrid Consistency

Each abstraction of non-atomic memory comes with its own implementation cost and its programming convenience, usually in a direct relationship. The solution of a programming task at hand

[2] Here tuple t is less than another tuple t' provided the pair (timestamp,counter) of t is less than that of t' in a lexicographic order.

may be optimized by mixing and matching different kinds of non-atomic memories or different kinds of consistency operations on the same memory. This realization prompted the definition of hybrid consistency where a set of *weak* and a set of *strong* consistency operations are defined for the same set of variables. A process interleaves a set of *strong* read/write operations with a set of *weak* read/write operations and the correctness of the memory has to be defined across both sets of operations. The particular choice of *strong* and *weak* consistency gives rise to different kinds of hybrid consistency. For example, in a definition due to Attiya and Friedman [5], *sequential consistency* [10] is adopted as the strong consistency condition and a weaker requirement in which program orderings need not be preserved is adopted as the weak consistency requirement. In fact, we have already seen a similar definition in PSO: program orderings are preserved across blocks separated by *stbar* commands whereas they are not preserved within the same block.

Here, we consider a definition of hybrid consistency that uses causal memory as the strong consistency condition and weak memory defined in Section 5 as the weak consistency condition. Essentially, the ordering between two *strong* operations is determined by causality, the ordering between a *weak* and a *strong* operation is determined by the program order, and no explicit ordering is introduced between two *weak* operations. Thus, in the absence of any *weak* operations, the semantics is exactly as for causal memory and in the absence of any *strong* operations, the semantics is exactly as for *weak* memory.

Similar to causal memory, we maintain a vector counter ts_p for each process p and sets X_w and X_r for each shared variable x. The translation rules for the *strong* operations are as in causal memory except that the local timestamps are incremented in the beginning so that orderings between *weak* and *strong* operations are preserved. For the *weak* operations, the actions are similar except that the local timestamp is not incremented. Formally, the rules of translation for process p are as follows.

1. A *strong* read statement $v := x$ is translated to

$$ts_p := nx_p(ts_p);$$
$$v, ts_p, X_r := m, nx_p(max(ts, ts_p)), X_r \cup \{(m, max(ts, ts_p))\} \text{ where } live(m, ts, x, p).$$

2. A *strong* write statement $x := m$ is translated to

$$ts_p := nx_p(ts_p);$$
$$X_w, ts_p := X_w \cup \{(m, ts_p)\}, nx_p(ts_p).$$

3. A *weak* read statement $v := x$ is translated to

$$v, X_r := m, X_r \cup \{(m, max(ts, ts_p))\} \text{ where } live(x, m, ts, p).$$

4. A *weak* write statement $x := m$ is translated to

$$X_w := X_w \cup \{(m, ts_p)\}.$$

Note that in the absence of any *strong* operations, the timestamp of a process is not incremented and therefore every value in X_w is live. In the absence of any *weak* operations, the timestamps are incremented before and after every operation. The net effect is the same as in causal memory, except that the timestamps grow twice as fast.

7 Discussion

We have presented a technique to define non-atomic memories in terms of atomic operations. Our approach is useful for the understanding and comparison of non-atomic memories and for the verification of parallel algorithms that use these memories. In a related work, Gibbons and Merritt [8] outline a framework for specifying non-blocking shared memories based on I/O automata [13] and use it to specify *release consistency* [7]. In the future, we intend to develop specifications for other existing consistency conditions as well as investigate semantic constraints under which atomic memory may be replaced by non-atomic memory in a program without affecting the final state of the computation.

References

[1] M. Ahamad, J. E. Burns, P. W. Hutto, and G. Neiger. Causal memory. In *Proceedings of the Fifth International Workshop on Distributed Algorithms, Lecture Notes in Computer Science.* Springer-Verlag, Berlin, October 1991.

[2] M. Ahamad, P. W. Hutto, and R. John. Implementing and programming causal distributed shared memory. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 274–281, 1991.

[3] J. H. Anderson and M. G. Gouda. Atomic semantics of nonatomic programs. *Information Processing Letters*, 28(2):99–103, 1988.

[4] T. E. Anderson. The performance of spin-lock alternatives for shared-memory multiprocessors. IEEE *Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[5] H. Attiya and R. Friedman. A correctness condition for high performance multiprocessors. In *Proceedings of the 24th Annual* ACM *Symposium on the the Theory of Computing*, pages 679–690, 1992.

[6] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison Wesley, Reading, Mass., 1988.

[7] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.

[8] P. B. Gibbons and M. Merritt. Specifying nonblocking shared memories. In *Proceedings of the Fourth Annual* ACM *Symposium on Parallel Algorithms and Architectures*, 1992.

[9] Sparc International Inc. *The SPARC Architecture Manual.* Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[10] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE *Transactions on Computers*, 28(9):690–691, Sept. 1979.

[11] L. Lamport. On interprocess communication, parts I and II. *Distributed Computing*, 1(2):77–101, 1986.

[12] R. Lipton and J. Sandberg. PRAM: a scalable shared memory. Technical Report CS-TR-180-88, Princeton University, 1988.

[13] N. A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual* ACM *Symposium on Principles of Distributed Computing*, pages 137–151, 1987.

[14] Z. Manna and A. Pnueli. How to cook a temporal proof system for your pet language. In *Proceedings of the 9th* ACM *Symposium on Principles of Programming Languages*, pages 141–154, 1983.

[15] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual* IEEE *Symposium on Foundations of Computer Science*, pages 46–57, 1977.