

# Can Expensive Synchronization be Avoided in Weak Memory Models?

## (Brief Announcement)

Lisa Higham\* and Jalal Kawash\*\*

Department of Computer Science, The University of Calgary, Canada, T2N 1N4  
Fax: +1 (403) 284 4707, Phone: +1 (403) 220 7696, 220 7681  
<http://www.cpsc.ucalgary.ca/{higham|kawash}>  
{higham|kawash}@cpsc.ucalgary.ca

**Abstract.** Process coordination problems have been extensively addressed in the context of sequential consistency. However, modern shared-memory systems present a large variety of ordering constraints on memory accesses that are much weaker than sequential consistency. We re-addressed two fundamental process coordination problems in the context of weak memory models. We proved that many models cannot support a solution to the critical section problem without additional powerful synchronization primitives. Nevertheless, we showed that certain versions of the producer/consumer problem can be solved even in the weakest models without the need for such powerful instructions. These instructions are expensive, and avoiding their use is desirable for better performance. This brief announcement groups and re-states some of our earlier results [11, 12, 15].

## 1 Introduction

Modern shared-memory systems present a large variety of memory models that are substantially weaker than sequential consistency [16]. These relaxed (or weak) memory consistency models increase the challenge of solving various coordination problems, because the relaxation of ordering constraints on memory accesses in these models makes reasoning about concurrency a subtle and involved issue. Therefore, we re-address fundamental process coordination problems in the context of weak memory consistency models.

Algorithms that coordinate processes via critical sections have long been known for sequentially consistent systems [22, 17, 20, 21, 4]. In fact, as shown by Lamport [17], even single-reader single-writer safe bits suffice to solve the critical section problem, as long as accesses to these seemingly weak objects are guaranteed to be sequentially consistent. In many weaker memory models, lack of one system-wide view of the ordering of reads and writes makes it impossible to construct critical sections using only read

---

\* Supported in part by Natural Sciences and Engineering Research Council of Canada research grant OGP004190.

\*\* Supported in part by a Natural Sciences and Engineering Research Council of Canada scholarship and an Izaak Walton Killam memorial scholarship.

and write operations. Therefore, shared-memory systems come equipped with additional basic operations that make synchronization possible. The use of synchronization primitives, however, incurs substantial additional execution time. So we are motivated to keep the use of strong primitives to a minimum. Hence we need to determine what kinds of coordination problems can be solved on common memory models using just read and write operations.

Specifically, this paper determines the possibility or impossibility of solving the critical section problem and versions of the producer/consumer [4] problem without using strong synchronization primitives on SPARC multiprocessors with memory models Total Store Ordering (TSO) and Partial Store Ordering (PSO) [23, 13, 15], Java Consistency [18, 8, 12], Coherence [5], Pipelined-RAM (P-RAM) [19], Goodman’s Processor Consistency (PC-G) [9], Causal Consistency (CC) [2], and Weak Ordering (WO) [5].

## 2 Memory Consistency Models

### 2.1 Basic Formalism

We briefly describe the formal framework here; a comprehensive treatment is elsewhere [15].

We model a multiprocess system as a collection of processes operating on a collection of shared data objects. We define a data object by providing the set of all sequences of allowable operations together with their results (similar to that of Herlihy and Wing [10].) Specifically, an *operation* is a 4-tuple (ac, obj, in, out) where “ac” is an action, “obj” is an object name, and “in” and “out” are sequences of parameters. The operation (ac, obj, in, out) means that the action “ac” with input parameters “in” is applied to the object “obj” yielding the output parameters “out”. A (*sequential data*) *object* is a specification of a set of sequences of operations. An arbitrary sequence of operations is *valid for object x* if and only if it is in the specification of *x*.

An operation  $o = (ac, obj, in, out)$  can be decomposed into the two *matching* components, (ac, obj, in), called the *operation-invocation* and denoted  $invoc(o)$ , and (ac, obj, out), called the matching *operation-response* and denoted  $resp(o)$ . Let  $(e_1, e_2, \dots)$  be a sequence consisting of operation-invocations and operation-responses. Then  $e_j$  follows  $e_i$  if and only if  $i < j$  and  $e_j$  immediately follows  $e_i$  if and only if  $i = j - 1$ .

Informally, a process interacts with data objects by issuing a stream of invocations to some subset of them and receiving a stream of responses that are interleaved with its invocations. This is formalized as follows. A *process* is a sequence of operation-invocations. A *process execution* is a sequence of operation-invocations and operation-responses such that each response follows its matching invocation. A *process computation* is the sequence of operations created from a process execution by augmenting each invocation in the process sequence with its matching response. Note that operations in a process computation maintain the order of their invocation components in the process execution.

A (*multiprocess*) *system*,  $(P, J)$ , is a collection  $P$  of processes and a collection  $J$  of objects, such that each operation-invocation of each process in  $P$  is applied to an object in  $J$ . A (*multiprocess*) *system execution* for a system  $(P, J)$ , is a collection of process

executions, one for each process in  $P$ . Similarly, a (*multiprocess*) *system computation* is a collection of process computations, one for each  $p$  in  $P$ .

Let  $(P, J)$  be a multiprocess system, and  $O$  be all the operations in a computation of this system.  $O|_p$  denotes all the operations that are in the process computation of  $p$  in  $P$ .  $O|x$  denotes all the operations that are applied to object  $x$  in  $J$ .

For the definition of some memory consistency models it is necessary to distinguish the operations that change a shared object from those that only inspect a shared object. Let  $O|_w$  denote that subset of  $O$  consisting of those operations in  $O$  that update (write) a shared object, and  $O|_r$  denote that subset consisting of the operations that only inspect (read) a shared object.

A (strict) partial order is an antireflexive, transitive relation on a set.<sup>1</sup> Given any collection of operations  $O$  on a set of objects  $J$ , a *linearization of  $O$*  is a (strict) linear order<sup>2</sup>  $(O, \xrightarrow{L})$  such that for each object  $x$  in  $J$ , the subsequence  $(O|x, \xrightarrow{L})$  of  $(O, \xrightarrow{L})$  is valid for  $x$ .

A (*memory*) *consistency model* is a set of constraints on system computations. A finite system computation of  $(P, J)$  *satisfies* consistency model  $D$  if the computation meets all the constraints in  $D$ . For any system  $(P, J)$  with only finite processes,  $(P, J)$  *delivers* memory consistency  $D$  if every computation that can arise from it satisfies  $D$ .

## 2.2 Useful Partial Orders

Five partial orders on operations in a system will be used in the memory consistency definitions below.

**program order:** Define *program order*, denoted  $(O, \xrightarrow{prog})$ , by  $o_1 \xrightarrow{prog} o_2$  if and only if  $\text{invoc}(o_2)$  follows  $\text{invoc}(o_1)$  in the definition of  $p$ , for some process  $p \in P$ .

**weak program order:** Define *weak program order*, denoted  $(O, \xrightarrow{wpo})$ , by  $o_1 \xrightarrow{wpo} o_2$  if  $o_1 \xrightarrow{prog} o_2$  and  $o_1$  and  $o_2$  are applied to the same object.

**write-before-read order:** Define *write-before-read order*, denoted  $(O, \xrightarrow{wbr})$ , by  $o_1 \xrightarrow{wbr} o_2$  if  $o_1$  is a write and  $o_2$  is a read, and  $o_2$  reads the value written by  $o_1$ .

**causal order:** Define *causal order*, denoted  $(O, \xrightarrow{causal})$ , by the irreflexive transitive closure of the union of  $(O, \xrightarrow{prog})$  and  $(O, \xrightarrow{wbr})$ . Namely,  $(O, \xrightarrow{causal}) = ((O, \xrightarrow{prog}) \cup (O, \xrightarrow{wbr}))^+$ .

**Java program order:** Define *Java program order* [7], denoted  $(O, \xrightarrow{jpo})$ , by  $o_1 \xrightarrow{jpo} o_2$  if  $o_1 \xrightarrow{prog} o_2$  and one of the following holds:

<sup>1</sup> Whenever we use the term partial order, we mean a strict partial order. A partial order is denoted by a pair  $(S, R)$  where  $S$  is a set and  $R \subset S \times S$ . The notation  $s_1 R s_2$  means  $(s_1, s_2) \in R$ . When the set  $S$  is understood,  $R$  denotes the partial order. If  $S' \subset S$  then  $(S', R)$  denotes the relation  $(S, R) \cap (S' \times S')$ .

<sup>2</sup> A strict linear order is a strict partial order  $(S, R)$  such that  $\forall x, y \in S \ x \neq y$ , either  $x R y$  or  $y R x$ . We also use linear order to mean a strict linear order.

1.  $o_1$  and  $o_2$  are applied to the same object,
2.  $o_1$  is a read and  $o_2$  is a write, and  $o' \xrightarrow{wbr} o_1$  where  $o'$  is a write by a process different from that of  $o_1$  (or  $o_2$ ), or
3. there exists  $o'$  such that  $o_1 \xrightarrow{jpo} o'$  and  $o' \xrightarrow{jpo} o_2$ .

### 2.3 Memory Consistency Definitions

In this section, we restate the definitions for the memory consistency models used in this paper. The definitions of WO, Java, and SPARC TSO and PSO below are specialized to ordinary read/write operations. For detailed development and for the general definitions, readers are referred to earlier work [15, 14, 12].

Let  $O$  be all the operations of a computation  $C$  of the multiprocess system  $(P, J)$ .

- $C$  satisfies *Sequentially Consistency* (abbreviated SC) [16] if there is a linearization  $(O, \xrightarrow{L})$  such that  $(O, \xrightarrow{prog}) \subseteq (O, \xrightarrow{L})$ .
- $C$  satisfies *Coherence* [5] if for each object  $x \in J$  there is a linearization  $(O|x, \xrightarrow{L_x})$  such that  $(O|x, \xrightarrow{prog}) \subseteq (O|x, \xrightarrow{L_x})$ .
- $C$  satisfies *Pipelined-RAM* (abbreviated P-RAM) [19] if for each process  $p \in P$  there is a linearization  $(O|p \cup O|w, \xrightarrow{L_p})$  such that  $(O|p \cup O|w, \xrightarrow{prog}) \subseteq (O|p \cup O|w, \xrightarrow{L_p})$ .
- $C$  satisfies *Processor Consistency–Goodman’s version* (abbreviated PC-G) [9] if for each process  $p \in P$  there is a linearization  $(O|p \cup O|w, \xrightarrow{L_p})$  such that
  1.  $(O|p \cup O|w, \xrightarrow{prog}) \subseteq (O|p \cup O|w, \xrightarrow{L_p})$ , and
  2.  $\forall q \in P$  and  $\forall x \in J$ ,  $(O|w|x, \xrightarrow{L_p}) = (O|w|x, \xrightarrow{L_q})$ .
- $C$  satisfies *Causal Consistency* (abbreviated CC) [2] if for each process  $p \in P$  there is a linearization  $(O|p \cup O|w, \xrightarrow{L_p})$  such that  $(O|p \cup O|w, \xrightarrow{causal}) \subseteq (O|p \cup O|w, \xrightarrow{L_p})$ .
- $C$  satisfies *Weak Ordering* (abbreviated WO) [5] if for each process  $p \in P$  there is some linearization  $(O|p \cup O|w, \xrightarrow{L_p})$  such that  $(O|p \cup O|w, \xrightarrow{wpo}) \subseteq (O|p \cup O|w, \xrightarrow{L_p})$ .
- $C$  satisfies *SPARC Total Store Ordering* (abbreviated TSO) [14, 15] if there exists a total order  $(O|w, \xrightarrow{writes})$  such that  $(O|w, \xrightarrow{prog}) \subseteq (O|w, \xrightarrow{writes})$  and  $\forall p \in P$  there is a total order  $(O|p \uplus O|w, \xrightarrow{merge_p})^3$ , such that:
  1.  $(O|p, \xrightarrow{prog}) = (O|p, \xrightarrow{merge_p})$ ,
  2.  $(O|w, \xrightarrow{writes}) = (O|w, \xrightarrow{merge_p})$ ,
  3. if  $w \in (O|p|w)$  then  $w_{O|p} \xrightarrow{merge_p} w_{O|w}$ , and
  4.  $((O|p \uplus O|w) \setminus (O_{inv_p} \cup O_{mw_p}), \xrightarrow{merge_p})$  is a linearization, where
 
$$O_{inv_p} = \{w \mid w \in (O|w \setminus O|p)|x \wedge \exists w' \in O|p|w|x \wedge w'_{O|p} \xrightarrow{merge_p} w \xrightarrow{merge_p} w'_{O|w}\}$$

$$O_{mw_p} = \{w_{O|w} \mid w \in O|p|w\}.$$

<sup>3</sup>  $A \uplus B$  denotes the disjoint union of  $A$  and  $B$ . If  $x \in A \cap B$ , then the copy of  $x$  in  $A$  (respectively  $B$ ) is denoted  $x_A$  (respectively  $x_B$ ).

- $C$  satisfies *SPARC Partial Store Ordering* (abbreviated PSO) [14, 15] if there exists a total order  $(O|w, \xrightarrow{\text{writes}})$  such that  $\forall x, (O|w|x, \xrightarrow{\text{prog}}) \subseteq (O|w|x, \xrightarrow{\text{writes}})$  and  $\forall p \in P$  there is a total order  $(O|p \uplus O|w, \xrightarrow{\text{merge}_p})$ , such that conditions 1 through 4 of TSO are satisfied.
- $C$  satisfies *Java Consistency* (abbreviated Java) [12, 15] if there is some total order  $(O|w, \xrightarrow{\text{writes}})$  such that,  $\forall p \in P$ :
  1. there is a linearization  $(O|p \cup O_{\text{vis}_p}, \xrightarrow{L_p})$  such that  $(O|p \cup O_{\text{vis}_p}, \xrightarrow{jpo}) \subseteq (O|p \cup O_{\text{vis}_p}, \xrightarrow{L_p})$ ,
  2.  $(O_{\text{vis}_p}, \xrightarrow{\text{writes}}) = (O_{\text{vis}_p}, \xrightarrow{L_p})$ , where
 
$$O_{\text{vis}_p} = \{w | w \in O|x \text{ for some } x \in J \wedge \exists r \in O|p \wedge w \xrightarrow{\text{wbr}} r \text{ or } \exists w' \in O|x \wedge w \xrightarrow{\text{writes}} w' \wedge w' \xrightarrow{\text{wbr}} r\}.$$

### 3 Critical Section Problem

In the critical section problem (CSP)[22], a set of processes coordinate to share a resource. We investigate minimum requirements for memory models to be able to provide a solution to CSP without the use of strong synchronization primitives.

A solution<sup>4</sup> to CSP must satisfy the safety property, mutual exclusion, and the progress property, freedom from deadlock. Typically, a fairness property is also required. Let  $\text{CSP}(n)$  denote a CSP problem with  $n$  processes.

**Theorem 1.** *There does not exist an algorithm that solves  $\text{CSP}(n)$ , even for  $n = 2$ , for:*

1. any system that delivers SPARC TSO (without the use of swap-atomic instructions).
2. any system that delivers SPARC PSO (without the use of store-barrier instructions).
3. any system that delivers Java (without the use of volatile variables or synchronized constructs).
4. any system that delivers PC-G (without the use of multi-writer objects).
5. any system that delivers P-RAM and Coherence but not PC-G.<sup>5</sup>
6. any system that delivers CC.
7. any system that delivers WO.

The systems listed in Theorem 1 cannot even support an unfair solution for CSP. The proofs appeared in [11, 12, 15].

An observation follows after recalling that Peterson’s Algorithm [21], which uses multi-writer registers, is a correct solution for the CSP even for PC-G [1]. However, Theorem 1(4) establishes that CSP is impossible in PC-G with single-writer objects.

**Corollary 1.** *In a system that delivers PC-G, multi-writer objects cannot be implemented from single-writer objects.*

For the rest of the systems in Theorem 1, the impossibilities apply even to multi-writer objects.

<sup>4</sup> The solutions referred to in this paper are non-cooperative [3].

<sup>5</sup> A system that is both P-RAM and Coherent is not necessarily PC-G [14].

## 4 Producer/Consumer Problem

Producer/Consumer [4] objects are frequently used for process coordination. The producer is a process that produces items and places them in a shared structure. A consumer is a process that consumes these items by removing them from the structure. We distinguish two structures with different solution requirements: the *set* structure where the order of consumption is insignificant, and the *queue* structure where items are consumed in the same order as that in which they were produced.

We denote the producer/consumer queue problem as  $P_mC_n$ -queue where  $m$  and  $n$  are respectively the number of producer and consumer processes. Similarly the producer/consumer set problem is denoted  $P_mC_n$ -set.

A solution to  $P_mC_n$ -set must satisfy the safety property that guarantees a one-to-one correspondence between produced and consumed items, and the progress property, freedom from deadlock.

A solution to  $P_mC_n$ -queue must satisfy the safety and progress properties in addition to the order property, which requires some total order on items that preserves both production and consumption orders. The *production order* is given by: if the production of item  $i$  completes before the production of item  $j$  begins, then  $i$  production order precedes  $j$ . The *consumption order* is defined similarly.

### 4.1 Queue Problem

Figure 1 shows a  $P_1C_1$ -queue algorithm that uses only single-writer objects,  $\mathcal{A}$ . Let an item  $it$  be composed of  $k$  bits,  $it$  is encoded such that the actual data is composed of  $k - 1$  bits. The  $k$ th bit is used for implicit synchronization. So initially, items are initialized to  $\perp$  concatenated with bit 0 (concatenation is denoted by  $||$ ). We also denote the complement of bit  $b$  as  $\bar{b}$ .

Proofs for the following appeared elsewhere [11, 12, 15].

**Theorem 2.**  $\mathcal{A}$  solves  $P_1C_1$ -queue for:

1. any system that delivers SPARC TSO (without the use of swap-atomic instructions).
2. any system that delivers SPARC PSO (without the use of store-barrier instructions).
3. any system that delivers PC-G (without the use of multi-writer objects).
4. any system that delivers P-RAM and Coherence but not PC-G.
5. any system that delivers CC.
6. any system that delivers WO.

Whenever  $P_1C_1$ -queue (and hence  $P_1C_1$ -set) can be solved using only single-writer objects, it can be solved using multi-writer objects. A solution to  $P_mC_n$ -queue can be constructed from a solution to CSP by protecting the queue structure in a critical section, so that it is accessed by only one process at a time. Hence, the  $P_mC_n$ -queue problem can be solved in SC and PC-G (with multi-writers) systems.

**Lemma 1.** There does not exist an algorithm that solves  $P_1C_n$ -queue or  $P_mC_1$ -queue, even for  $n = 2$  or  $m = 2$ , for each of the systems of Theorem 1.

**Theorem 3.** There does not exist an algorithm that solves  $P_mC_n$ -queue for each of the systems of Theorem 1 for  $m + n \geq 3$ .

---

```

shared var:
P: array[0..n-1] of item (initialized to ( $\perp$  || 0))
C: array[0..n-1] of item (initialized to ( $\perp$  || 0))
define function b(it:item): returns the synchronization bit in it
define function v(it:item): returns the data bits in it

producer:
var
  in : 0..n-1
  itp : item
  bitp : bit

in ← 0
repeat
  ... produce itp
  bitp ← b(P[in])
  while b(C[in]) ≠ bitp do nothing
  P[in] ← v(itp) ||  $\overline{\text{bit}_p}$ 
  in ← in + 1 mod n
until false

consumer:
var
  out : 0..n-1
  itc : item
  bitc : bit

out ← 0
repeat
  bitc ← b(C[out])
  while b(itc ← P[out]) = bitc do nothing
  C[out] ← v(itc) ||  $\overline{\text{bit}_c}$ 
  ... consume itc
  out ← out + 1 mod n
until false

```

**Fig. 1.**  $\mathcal{A}$ , a  $P_1C_1$ -queue algorithm

---

## 4.2 Set Problem

Although the  $P_mC_n$ -queue cannot be solved in many weak systems, we have shown that the  $P_mC_n$ -set can be solved in many of those systems [11, 12, 15].

**Corollary 2.** *There exists an algorithm that solves  $P_mC_n$ -set for each of the systems of Theorem 2.*

**Theorem 4.** *There does not exist an algorithm that solves  $P_mC_n$ -set, even for  $n = 1$  and  $m = 1$ , for any system that delivers Java (without the use of volatile variables or synchronized constructs).*

## 4.3 Summary

Table 1 summarizes our impossibility and possibility results.

**Table 1.** Summary of possibilities ( $\checkmark$ ) and impossibilities ( $\otimes$ )

	CSP	$P_1C_1$ -queue	$P_1C_n$ -queue	$P_mC_1$ -queue	$P_mC_n$ -queue	$P_mC_n$ -set
Java with only read/write operations	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$
SPARC TSO or PSO with only read/write operations	$\otimes$	$\checkmark$	$\otimes$	$\otimes$	$\otimes$	$\checkmark$
P-RAM & Coherence but not PC-G	$\otimes$	$\checkmark$	$\otimes$	$\otimes$	$\otimes$	$\checkmark$
CC	$\otimes$	$\checkmark$	$\otimes$	$\otimes$	$\otimes$	$\checkmark$
WO with only read/write operations	$\otimes$	$\checkmark$	$\otimes$	$\otimes$	$\otimes$	$\checkmark$
PC-G with only single-writers	$\otimes$	$\checkmark$	$\otimes$	$\otimes$	$\otimes$	$\checkmark$
PC-G with multi-writers [1]	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
SC	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$

## Acknowledgement

We thank an anonymous referee for a helpful comment regarding single-writer single-reader safe bits.

## References

1. Mustaque Ahamad, Rida Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 251–260, June 1993. Technical Report GIT-CC-92/34, College of Computing, Georgia Institute of Technology.

2. Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: Definitions, implementations, and programming. *Distributed Computing*, 9:37–49, 1995.
3. Hagit Attiya and Roy Friedman. A correctness condition for high performance multiprocessors. *SIAM Journal of Computing*, 27(6):1637–1670, 1998.
4. E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965. Reprinted in [6].
5. Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. *Proc. of the 13th Int’l Symp. on Computer Architecture*, pages 434–442, June 1986.
6. F. Genuys, editor. *Programming Languages*. Academic Press, 1968.
7. Alex Gontmakher, Ayal Itskovitz, and Assaf Schuster. Java consistency: Non-operational characterizations of Java memory behavior. Technical Report CS0922, Computer Science Department, Technion, November 1997.
8. Alex Gontmakher and Assaf Schuster. Characterizations of Java memory behavior. In *Proc. of the 12th Int’l Parallel Processing Symp.*, April 1998.
9. James Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, March 1989.
10. Maurice Herlihy and Jeannette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.
11. Lisa Higham and Jalal Kawash. Critical sections and producer/consumer queues in weak memory systems. In *Proc. of the 1997 Int’l Symp. on Parallel Architectures, Algorithms, and Networks*, pages 56–63, December 1997.
12. Lisa Higham and Jalal Kawash. Java: Memory consistency and process coordination (extended abstract). In *Proc. of the 12th Int’l Symp. on Distributed Computing*, pages 201–215, September 1998.
13. Lisa Higham, Jalal Kawash, and Nathaly Verwaal. Defining and comparing memory consistency models. In *Proc. of the 10th Int’l Conf. on Parallel and Distributed Computing Systems*, pages 349–356, October 1997.
14. Lisa Higham, Jalal Kawash, and Nathaly Verwaal. Weak memory consistency models part I: Definitions and comparisons. Technical Report 98/612/03, Department of Computer Science, The University of Calgary, January 1998.
15. Jalal Kawash. Process coordination issues in systems with weak memory consistency. PhD Dissertation draft, Department of Computer Science, The University of Calgary.
16. Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.
17. Leslie Lamport. The mutual exclusion problem (parts I and II). *Journal of the ACM*, 33(2):313–326 and 327–348, April 1986.
18. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
19. Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report 180-88, Department of Computer Science, Princeton University, September 1988.
20. Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
21. Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.
22. Michel Raynal. *Algorithms for Mutual Exclusion*. The MIT Press, 1986.
23. David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual version 9*. Prentice-Hall, 1994.