Weak Memory Consistency Models
Part II: Process Coordination Problems*†

Lisa Higham, Jalal Kawash, and Nathaly Verwaal
Department of Computer Science, The University of Calgary, Alberta, Canada T2N 1N4
email:<last name>@cpsc.ucalgary.ca

Abstract

Process coordination problems have been extensively addressed in the context of sequential consistency. However, modern multiprocessors present a large variety of memory models that are anything but sequentially consistent. In these machines, the ordering constraints on memory accesses are few. We re-address two fundamental process coordination problems in the context of weak memory models. We prove that many models cannot support a solution to the critical section problem without additional powerful synchronization primitives. Nevertheless, we show 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.

1 Introduction

We readdress fundamental process coordination problems in the context of weak memory consistency models. These problems have been extensively studied in the context of sequential consistency [17]. However, modern multiprocessors present a large variety of memory models that are anything but sequentially consistent. Relaxed memory consistency models increase the challenge of solving various coordination problems. The relaxation of ordering constraints on memory accesses in these models makes reasoning about concurrency a subtle and involved issue.

Algorithms that coordinate processes via critical sections have long been known for sequentially consistent systems [23, 18, 21, 22, 6]. However, many weaker memory models cannot provide this coordination through only read and write operations. Therefore, multiprocessors come equipped with additional basic operations which 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.

In a companion paper, we define a formal framework for specifying memory consistency models [14]. In this paper, we use that formalism to address fundamental process coordination problems for some memory consistency models where constraints on orderings of memory accesses are weaker than those guaranteed by sequential consistency. Specifically, this paper determines the possibility or impossibility of solving the critical section problem and versions of the producer/consumer [6] problem without using strong synchronization primitives on SPARC multiprocessors (with memory models Total Store Order and Partial Store Order)[24] and on common weak memory models from the literature (including coherence [12], Pipelined-RAM [20], Goodman's processor consistency [12]).

Investigating process coordination problems in weak memory models was triggered by the Georgia Tech. team [4] when they proved that Peterson's mutual exclusion algorithm [22] is correct for Proces-

*This research was supported by a trust fund and a post-graduate scholarship from the Natural Sciences and Engineering Research Council of Canada and by a research grant from The University of Calgary.

†Parts of this paper appeared earlier in a conference paper [13].

sor Consistency–Goodman's definition (PC-G) and Lamport's Bakery algorithm [16] is not. In this paper, we give a more general result. We prove that it is impossible to solve mutual exclusion in a PC-G system with only single-writer objects.

Leslie Lamport proposed an interesting method to derive synchronization primitives from the correctness proof of an algorithm [19]. However, the method is not optimal in the sense that it may derive more synchronization than required.

The team at Stanford investigated constrained programming styles or techniques for labeling memory accesses so that proper synchronization is guaranteed [10, 9, 1, 11]. In particular, the notion of properly labeled programs (PLP) emerged. Memory accesses are classified as *ordinary* or *special*. Special accesses are further classified as *competing* and *non-competing*. Further classifications were also proposed. The point behind PLP is to label all accesses in a program according to their actual categories. This will allow the underlying architecture, for example, to execute special instructions in a constrained manner, while ordinary instructions may be reordered in ways that best suit performance optimization. The labeling is to be performed by the programmer and to be exploited by the compiler for automatic synchronization generation.

Another programmer-oriented approach was developed at Wisconsin-Madison [1, 2, 3], the Sequential Consistency Normal Form (SCNF). Their notion of Data Race Free (DRF) programs is very similar to that of PLP[1]. This approach develops a programming style and then suggests how to tailor the hardware to support it. Attiya, Chaudhuri, and Friedman have taken a complementary approach, which first specifies the memory consistency model and then develops a programming style [5].

PLP and DRF models are interesting approaches. However, unfortunately, not all information is available to the programmer statically. Most of the time, programmers have to make conservative decisions which lead to unnecessary synchronization. Our approach gives programmers an insight of what problems can be solved without explicit synchronization (i.e., with only ordinary instructions).

Section 2 reviews the basic concepts of our framework and restates the definitions of the memory consistency models referred to in this paper. Section 3 examines the ability of these models to support mutual exclusion. Section 4 examines the ability of these models to solve some producer/consumer problems. Our contributions are summarized in Section 5.

2 Memory Consistency Models

In this section, we summarize the concepts used in our framework and define the memory consistency models referred to in subsequent sections. For a comprehensive presentation and justification of definitions, the reader is referred to the companion paper [14].

An *action* is a 4-tuple (op, obj, in, out) where "op" is an operation, "obj" is an object name, and "in" and "out" are sequences of parameters [2]. An action $o =$ (op, obj, in, out) can be decomposed into the two *matching* components, (op, obj, in), called the *action-invocation* and denoted invoc(o), and (op, obj, out), called the matching *action-response* and denoted resp(o). A *(sequential data) object* is specified by a set of sequences of actions. A sequence of actions is *valid for object* x if and only if it is in the specification of x.

A *process* is a sequence of action-invocations. A *process computation* is the sequence of actions created from a process execution by augmenting each invocation in the process sequence with its matching response. A *(multiprocess) system*, (P, J), is a collection P of processes and a collection J of objects, such that each action-invocation of each process p in P is applied to an object in J. A *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 actions in a computation of this system. $O|p$ denotes all the actions $o \in O$ that are in the process computation of p in P. $O|x$ are all the actions that are applied to object x in J. For the definition of some memory consistency models it is necessary to distinguish the actions that change (write) a shared object from those that only inspect (read) a shared object. Let O_w be that subset of O consisting of those actions in O that update a shared object, and O_r be that subset

[1]The two teams, in fact, joined their work together; see [9] for example.

[2]The action (op, obj, in, out) means that the operation "op" with input parameters "in" is applied to the object "obj" yielding the output parameters "out".

consisting of the actions that only inspect a shared object. There are also some consistency models that provide other classes of actions. These are defined as needed.

Given any collection of actions O on a set of objects J, a *linearization of* O is a linear order[3] $(O, <_L)$ such that for each object $x \in J$, the subsequence $(O|x, <_L)$ of $(O, <_L)$ is valid for x. Define the partial order $(O, \xrightarrow{prog})$, called the *program order*, as follows. Action o_1 *program-precedes* o_2, denoted $o_1 \xrightarrow{prog} o_2$, if and only if invoc$(o_2)$ follows invoc(o_1) in the definition of process p. Define *weak program order*, denoted $\xrightarrow{weak-prog}$, by $o_1 \xrightarrow{weak-prog} o_2$ if $o_1 \xrightarrow{prog} o_2$ and either 1) at least one of $\{o_1, o_2\}$ is a synchronization action, or 2) $\exists o'$ such that o' is a synchronization action and $o_1 \xrightarrow{prog} o' \xrightarrow{prog} o_2$, or 3) o_1 and o_2 are to the same object.

A *(memory) consistency condition* is a set of constraints on system computations. A computation C satisfies some consistency condition D if the computation meets all the constraints of D. A system provides memory consistency D if every computation that can arise from the system satisfies the consistency condition D. In the following definitions, A computation C satisfies some consistency condition D if the computation meets all the conditions of D. A system provides memory consistency D if every computation that can arise from the system satisfies the consistency condition D.

Following are the definitions for sequential consistency (SC) [17], coherence [12], Ahamad et al. 's interpretation of Pipelined-RAM (P-RAM-A) [20, 4], our interpretation of Pipelined-RAM (P-RAM) [20, 14], Goodman's processor consistency (PC-G) [12], weak ordering (WO) [7], coherent weak ordering (WO$_{coherent}$), SPARC total store ordering (TSO) and partial store ordering (PSO) [24].

Let O be all the actions of a computation C of the multiprocess system (P, J). Then C is **SC** if there is a linearization $(O, <_L)$ such that $(O, \xrightarrow{prog}) \subseteq (O, <_L)$.

Let O be all the actions of a computation C of the multiprocess system (P, J). Then E is **coherent** if for each object $x \in J$ there is some linearization $(O|x, <_{L_x})$ satisfying $(O|x, \xrightarrow{prog}) \subseteq (O|x, <_{L_x})$.

Let O be all the actions of a computation C of the multiprocess system (P, J). Then E is **P-RAM-A** if for each process $p \in P$ there is a linearization $(O|p \cup O_w, <_{L_p})$ satisfying $(O|p \cup O_w, \xrightarrow{prog}) \subseteq (O|p \cup O_w, <_{L_p})$.

Let O be all the actions of a computation C of the multiprocess system (P, J). Then C is **P-RAM** if for each process $p \in P$ there is a linearization $(O|p \cup O_w, <_{L_p})$ satisfying 1) $(O|p \cup O_w, \xrightarrow{prog}) \subseteq (O|p \cup O_w, <_{L_p})$, and 2) for any $m \geq 1$ and for all $w_{p_0}, w_{p_1}, \ldots, w_{p_m} \in O_w$, where w_{p_i} is a write by process $p_i \in P$, if $w_{p_0} <_{L_{p_1}} w_{p_1} <_{L_{p_2}} w_{p_2} <_{L_{p_3}} \ldots <_{L_{p_m}} w_{p_m}$ then $w_{p_0} <_{L_{p_0}} w_{p_m}$.

Let O be all the actions of a computation C of a multiprocess system (P, J). Then C is **PC-G** if for each process $p \in P$ there is a linearization $(O|p \cup O_w, <_{L_p})$ satisfying 1) $(O|p \cup O_w, \xrightarrow{prog}) \subseteq (O|p \cup O_w, <_{L_p})$, and 2) $\forall q \in P$ and $\forall x \in J$ $(O_w \cap O|x, <_{L_p}) = (O_w \cap O|x, <_{L_q})$.

Let O be all the actions of a computation C of a multiprocess system (P, J). Then C is **WO** if for each process $p \in P$ there is some linearization $(O|p \cup O_w, <_{L_p})$ satisfying 1) $(O|p \cup O_w, \xrightarrow{weak-prog}) \subseteq (O|p \cup O_w, <_{L_p})$, and 2) $\forall q \in P$ $(O_w \cap O_{synch}, <_{L_p}) = (O_w \cap O_{synch}, <_{L_q})$.

Let O be all the actions of a computation C of a multiprocess system (P, J). Then E is **WO**$_{coherent}$ if for each process $p \in P$ there is some linearization $(O|p \cup O_w, <_{L_p})$ satisfying the two conditions of WO and $\forall q \in P$ and $\forall x \in J$ $(O_w|x, <_{L_p}) = (O_w|x, <_{L_q})$.

In the following, $(A \uplus B)$ denotes the disjoint union of sets A and B, and if $x \in A \cap B$ then the copy of x in A is denoted x_A and the copy of x in B is denoted x_B. Let O_a denote the set of swap atomic actions and O_{sb} denote the set of store barrier actions provided by the SPARC architecture [24]. Let O_r denote the set of actions with read semantics. Then, $O_w \cap O_r = O_a$.

Let O be all the actions of a computation C of the multiprocess system (P, J). Then C is **TSO** 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 $(O|p \uplus O_w, \xrightarrow{merge_p})$, satisfying:

1. $(O|p, \xrightarrow{prog}) = (O|p, \xrightarrow{merge_p})$, and
2. $(O_w, \xrightarrow{writes}) = (O_w, \xrightarrow{merge_p})$, and
3. if $w \in (O|p \cap O_w)$ then $w_{O|p} \xrightarrow{merge_p} w_{O_w}$, and

[3] A linear order is an irreflexive partial order (S, R) such that $\forall x, y \in S$ $x \neq y$, either xRy or yRx.

4. $((O|p \uplus O_w)\backslash(O_{invisible_p} \cup O_{memwrites_p}), \stackrel{merge_p}{\longrightarrow})$ is a linearization, where
$O_{invisible_p} = \{w \mid w \in (O_w \backslash O|p) \cap O|x \wedge \exists w' \in O|x \cap O|p \cap O_w \wedge w'_{O|p} \stackrel{merge_p}{\longrightarrow} w \stackrel{merge_p}{\longrightarrow} w'_{O_w}\}$
$O_{memwrites_p} = \{w_{O_w} \mid w \in O|p \cap O_w\}$, and

5. let $w \in (O|p \cap O_w)$ and $a \in (O|p \cap O_a)$, if $w \stackrel{prog}{\longrightarrow} a$, then $w_{O_w} \stackrel{merge_p}{\longrightarrow} a$, and if $a \stackrel{prog}{\longrightarrow} w$, then $a \stackrel{merge_p}{\longrightarrow} w_{O|p}$

Let O be all the actions resulting from a computation C of the multiprocess system (P, J). Then C is **PSO** if there exists a total order $(O_w, \stackrel{writes}{\longrightarrow})$ such that $\forall x$, $(O_w \cap O|x, \stackrel{prog}{\longrightarrow}) \subseteq (O_w \cap O|x, \stackrel{writes}{\longrightarrow})$ and $\forall p \in P$ there is a total order $(O|p \uplus O_w, \stackrel{merge_p}{\longrightarrow})$, satisfying items 1 through 4 of TSO and (5) if $sb \in (O|p \cap O_{sb})$ and $w, u \in (O|p \cap O_w)$ and $w \stackrel{prog}{\longrightarrow} sb \stackrel{prog}{\longrightarrow} u$, then $w_{O_w} \stackrel{merge_p}{\longrightarrow} u_{O_w}$.

These definitions are given assuming that the multiprocess system terminates. They can be extended so that they apply to long-lived systems. Let O be all actions of some finite computation C of a system (P, J). To establish that C satisfies consistency condition D, we must provide a set of sequences $\mathcal{S}$, each composed of actions in O, that satisfy the constraints of D. Call such an $\mathcal{S}$ a set of *satisfying sequences* of (C, D). A sequence s *extends* $\hat{s}$ if $\hat{s}$ is a prefix of s. Let $\{C_p\}$ and $\{\hat{C}_p\}$ be sets of process computations. A computation $C = \bigcup_{p \in P}\{C_p\}$ *extends* $\hat{C} = \bigcup_{p \in P}\{\hat{C}_p\}$ if $\forall p \in P$, C_p extends $\hat{C}_p$.

An *infinite* computation C satisfies consistency condition D if $\forall p \in P$ and for every finite prefix $\hat{C}_p$ of C_p, there is a finite prefix $\hat{C}_q$ of C_q $\forall q \neq p$, such that $\hat{C} = \bigcup_{p \in P}\{\hat{C}_p\}$ satisfies D. Furthermore, for any finite $\tilde{C} = \bigcup_{p \in P}\{\tilde{C}_p\}$ that (1) extends $\hat{C}$ and (2) satisfies D and (3) $\forall p \in P$, $\tilde{C}_p$ is a finite prefix of C_p, $\tilde{C}$ has satisfying sequences $\tilde{\mathcal{S}}$ such that there exists satisfying sequences $\hat{\mathcal{S}}$ for $(\hat{C}, D)$ and $\forall \tilde{s} \in \tilde{\mathcal{S}}$, $\tilde{s}$ extends a unique $\hat{s}$ in $\hat{\mathcal{S}}$.

We illustrate this via an example. Consider Computation 1 where the ∞ superscript indicates an infinite repetition of the action $r(x)0$.

Computation 1 $\left\{ \begin{array}{l} p : [r(x)0]^\infty \\ q : w(x)1 \end{array} \right.$

In this computation, for any finite prefix of C_q, there is a a finite prefix of C_p such that the resulting finite computation satisfies SC, say. In particular, C_p is $[r(x)0]^m$ where m is finite, and C_q is $w(x)1$. Moreover, $[r(x)0]^m\ w(x)1$ is a linearization satisfying SC. However, any finite prefix of this computation of size $n > m+1$ can not extend the above linearization and maintain validity at the same time. More precisely, $[r(x)0]^m\ w(x)1\ [r(x)0]^{n-m-1}$ is not SC. Thus, Computation 1 is not SC.

3 Critical Section Problem

In the critical section problem (CSP)[23], 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.

We assume each process has the following structure:

repeat
    <remainder>
    <entry>
    <critical section>
    <exit>
until *false*

Given the multiprocess system (P, J), a solution to the critical section problem must satisfy the following properties:

- **Mutual Exclusion:** At any time there is at most one $p \in P$ in its *<critical section>*.
- **Progress:** If at least on process is in *<entry>*, then eventually one will be in *<critical section>*.

- **Fairness:** If $p \in P$ is in <*entry*>, then p will eventually be in <*critical section*>.

Now, we turn to some impossibilities concerning CSP. First, we need the following lemma.

Lemma 3.1 *A P-RAM-A (and hence a P-RAM) system (P, J) with only single-writer objects satisfies PC-G.*

Proof: Let $<_{L_p}$ be a linearization for $p \in P$ of $(O|p \cup O_w)$ that is guaranteed by P-RAM-A. Similarly, let $<_{L_q}$ be a linearization for $q \in P$ of $(O|q \cup O_w)$. Since, for any object $x \in J$, there is only one processor, say r, that writes to x, and both $<_{L_p}$ and $<_{L_q}$ have all these writes to x in the program order of r, the order of the writes to x in $<_{L_p}$ is the same as the order of the writes to x in $<_{L_q}$. Therefore, the definition of PC-G is satisfied. ∎

We will use the partial computations 2, 3, and 4 defined as follows.

Computation 2 $\begin{cases} p : o_1^p, o_2^p, ..., o_k^p & (p \textit{ is in its } < \textit{critical section} >) \\ q : \lambda \end{cases}$

where λ denotes the empty sequence and o_i^p denotes the i^{th} operation of p.

Computation 3 $\begin{cases} p : \lambda \\ q : o_1^q, o_2^q, ..., o_l^q & (q \textit{ is in its } < \textit{critical section} >) \end{cases}$

Computation 4 $\begin{cases} p : o_1^p, o_2^p, ..., o_k^p & (p \textit{ is in its } < \textit{critical section} >) \\ q : o_1^q, o_2^q, ..., o_l^q & (q \textit{ is in its } < \textit{critical section} >) \end{cases}$

Claim 3.2 *There is no algorithm even for two processes that solves CSP in any of the following systems:*

1. *a PC-G system with only single-writer objects.*
2. *a system that is only P-RAM and coherent but not PC-G*[4]*.*
3. *a TSO system without the use of swap-atomic instructions, or a PSO system without the use of store-barrier instructions.*

Proof: For any of these systems, $(\{p, q\}, J)$, assume for the sake of contradiction that there is a mutual exclusion algorithm A that solves CSP. If A runs with p in $<$ *entry* $>$ and with q in $<$ *remainder* $>$, then by the Progress property, p must enter its $<$ *critical section* $>$ producing a partial computation of the form of Computation 2. Similarly, if A runs with q in $<$ *entry* $>$ and without p participating, a computation of the form of Computation 3 results. The existence of computations 2 and 3 is used to build a computation of the form of Computation 4. Then, this computation is shown to satisfy the definition of each one of these systems. Hence Computation 4 is a possible computation of A. However, in Computation 4 both p and q are in their critical sections simultaneously, contradicting the requirement that A satisfies Mutual Exclusion.

1. Let $(o_1^p, ..., o_k^p)((o_1^q, ..., o_l^q)|_w)$ [5] be a total order of $(O|p \cup O_w)$ and let $(o_1^q, ..., o_l^q)((o_1^p, ..., o_k^p)|_w)$ be a total order of $(O|q \cup O_w)$. Clearly each preserves $\xrightarrow{prog}$ and satisfies condition 2 of the P-RAM definition. Also, each is a linearization because the first part corresponds to a possible computation, and the second part contains only writes. Therefore, Computation 4 is P-RAM. However, by Lemma 3.1, Computation 4 is also PC-G if it uses only single-writer objects.

2. As shown in item 1, Computation 4 is P-RAM. To prove that Computation 4 is coherent, let o_j^p be the first write to object x by p. Then, $(o_1^p, ..., o_{j-1}^p)|_x \ (o_1^q, ..., o_l^q)|_x \ (o_j^p, ..., o_k^p)|_x$ is a total order on $O|x$ that clearly preserves $\xrightarrow{prog}$. Each of the segments $(o_1^p, ..., o_{j-1}^p)|_x$, $(o_j^p, ..., o_k^p)|_x$ and $(o_1^q, ..., o_l^q)|_x$ occur in a possible computation; the first two are concatenated before there has been any write to x by p and the first operation of the last segment is a write to x, which obliterates any previous change to x by q. Therefore the sequence is a linearization of $O|x$, showing that Computation 4 is coherent.

[4] A system that is both P-RAM and coherent is not necessarily PC-G [14].

[5] $(o_1^q, ..., o_l^q)|_w$ denotes the subsequence of $o_1^q, ..., o_l^q$ such that every o_i^q in the subsequence is in O_w. Similarly, $(o_1^q, ..., o_l^q)|_x$ denotes the subsequence of $o_1^q, ..., o_l^q$ such that every o_i^q in the subsequence is in $O|x$.

3. To see that Computation 4 satisfies TSO, we imagine a situation where p and q enter their critical sections before the contents of their write-buffers are committed to main memory. Specifically, let $(O_w, \stackrel{writes}{\longrightarrow})$ be $(o_1^p, ..., o_k^p, o_1^q, ..., o_l^q)|_w$ which clearly preserves $\stackrel{prog}{\longrightarrow}$. Let $\stackrel{merge_p}{\longrightarrow}$ be $o_1^p, ..., o_k^p$, $(o_1^p, ..., o_k^p$, $o_1^q, ..., o_l^q)|_w$, and $\stackrel{merge_q}{\longrightarrow}$ be $o_1^q, ..., o_l^q$, $(o_1^p, ..., o_k^p$, $o_1^q, ..., o_l^q)|_w$, conditions 1, 2, 3, and 5 of TSO are obviously satisfied. Condition 4, follows because, for p, $((O|p \uplus O_w)\backslash(O_{invisible_p} \cup O_{memwrites_p}) = o_1^p, ..., o_k^p$ which is clearly a linearization since it is Computation 2, and similarly for q.

■

Observe that none of the arguments in Claim 3.2 depends on the Fairness property, so the systems listed there cannot even support an unfair solution for CSP. A second observation follows after recalling that Peterson's Algorithm [22], which uses multi-writer registers, is a correct solution for the CSP even for PC-G [4]. However, Claim 3.2(1) establishes that CSP is impossible in PC-G with single-writer objects.

Corollary 3.3 *In a PC-G system, multi-writer objects cannot be implemented from single-writer objects.*

For the rest of the systems in Claim 3.2, the proof did not assume single-writer objects. Therefore, the impossibilities apply even to multi-writer objects.

If CSP cannot be solved in a model A, then it cannot be solved in any model that is strictly weaker than A. The following Corollary needs no proof.

Corollary 3.4 *There is no algorithm even for two processes that solves CSP in any of the following systems: P-RAM-A, WO with only ordinary actions, or* $WO_{coherent}$ *with only ordinary actions.*

4 Producer/Consumer Problem

Producer/Consumer [6] 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 which removes these items from the structure. We distinguish two structures whose solution requirements vary: 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.

Producers and consumers are assumed to have the following forms:

```
producer:                     consumer:
repeat                        repeat
      <entry>                       <entry>
      <producing>                   <consuming>
      <exit>                        <exit>
until false                   until false
```

Note that the form of the producer and consumer processes assumes infinite computations. 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 following:

- **Safety:** There is a one-to-one correspondence between produced and consumed items.

- **Progress:** If a producer (respectively consumer) is in *<entry>*, then it will eventually be in *<producing>* ((respectively *<consuming>*) and subsequently in *<exit>*.

A solution to P_mC_n-queue must satisfy Safety, Progress, and the following property:

- **Order:** consumers consume items in the same order as that in which the items were produced.

4.1 Queue Problem

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

We will prove that $\mathcal{ALG_S}$ is correct for any system that is only WO. Our proof assumes that the arrays `P` and `C` are a single cell each (`n = 1`). The extension to the general case is straightforward but more tedious. We refer to the producer and consumer processes by p and c respectively. To simplify notation, we will pay no attention to the $k-1$ data bits written or read in $\mathcal{ALG_S}$. What really matters is the value of the kth synchronizing bit. In the following, o_p^i indicates that action o is performed by p in its ith iteration. Let $r_p^i(\mathtt{P})b$ be p's read of line (1), and $w_p^i(\mathtt{P})\bar{b}$ the write of line (3). Similarly, $r_c^i(\mathtt{C})b$ and $w_c^i(\mathtt{C})\bar{b}$ are those of lines (4) and (6) respectively. Let $r_p^i(\mathtt{C})\bar{b}$ denote the last read in line (2) which caused p to break the while loop, call such a read a *successful read*. Similarly, let $r_c^i(\mathtt{P})b$ be c's successful read in line (5).

Lemma 4.1 *For any WO finite prefix of an infinite computation $C_\mathcal{A}$ of $\mathcal{ALG_S}$ in a WO system, any set of WO linearizations for p and c of the actions in $C_\mathcal{A}$ satisfies the following constraints $\forall i, j \geq 1$:*

1. $r_p^i(\mathtt{P})b <_{L_p} w_p^i(\mathtt{P})\bar{b} <_{L_p} r_p^{i+1}(\mathtt{P})\bar{b}$
2. $r_c^i(\mathtt{C})b <_{L_c} w_c^i(\mathtt{C})\bar{b} <_{L_c} r_c^{i+1}(\mathtt{C})\bar{b}$
3. $r_p^i(\mathtt{C})\bar{b} <_{L_p} w_p^i(\mathtt{P})\bar{b}$
4. $r_c^i(\mathtt{P})b <_{L_c} w_c^i(\mathtt{C})\bar{b}$
5. $r_p^i(\mathtt{C})b <_{L_p} w_c^j(\mathtt{C})\bar{b} <_{L_p} r_p^{i+1}(\mathtt{C})\bar{b}$
6. $r_c^i(\mathtt{P})b <_{L_c} w_p^j(\mathtt{P})\bar{b} <_{L_c} r_c^{i+1}(\mathtt{P})\bar{b}$
7. $w_p^1(\mathtt{P})1 <_{L_c} r_c^1(\mathtt{P})1$

Proof: Constraints 1 and 2 are imposed by the definition of WO, which guarantees that program order is maintained on per object basis.

Constraint 3 holds because individual processes are self-consistent (the result of an individual process execution, without any other intervention, is the same as if it executed sequentially). Therefore, p can not commit to $w_p^i(\mathtt{P})\bar{b}$ before a successful read of `C` in the while loop. The same argument applies to constraint 4.

The reads by p of `C` in iterations i and $i+1$ must return opposite values. This means that a write by c to `C` must intervene between them from p's point of view, thus establishing constraint 5. Constraint 6 follows by a similar argument. Finally, constraint 7 follows from the initial state of `P` and `C` and from $\mathcal{ALG_S}$ itself. ■

Claim 4.2 $\mathcal{ALG_S}$ *solves P_1C_1-queue in a system that is only WO.*

Proof: Let $C_\mathcal{A}$ be a WO finite prefix of an infinite computation of $\mathcal{ALG_S}$ in a WO system. We will show that any set of linearizations of the actions of $C_\mathcal{A}$ that satisfy WO will also satisfy Safety, Progress, and Order.

Let $C_\mathcal{A}$ contain n writes by p to `P`. If $n = 0$, then by the initial states of `P` and `C`, c must have performed 0 writes. If $n = 1$, then c must have performed 0 or 1 writes by Lemma 4.1 constraints 4 and 6. If $n > 1$, then there must be at least n successful reads by p of `C`. By Lemma 4.1 constraint 5, there are at least $n-1$ writes by c to `C`. But by constraints 6 and 7 of Lemma 4.1, there are at most n successful reads by c of `P` in $C_\mathcal{A}$. Thus, $C_\mathcal{A}$ has at most n writes by c (by Lemma 4.1 constraint 4). Therefore, $C_\mathcal{A}$ contains either n or $n-1$ writes by c to `C`.

By the pigeon hole principle, before each successful read by c of `P`, there is one of the n writes by p. Moreover, if there are only $n-1$ reads by c of `C`, $w_p^n(\mathtt{P})b$ is placed after $r_c^{n-1}(\mathtt{P})\bar{b}$.

Precisely, $<_{L_c} = w_p^1(\mathtt{P})1$, $r_c^1(\mathtt{P})|1$, $r_c^1(\mathtt{C})0$, $w_c^1(\mathtt{C})1$, $w_p^2(\mathtt{P})0$, $r_c^2(\mathtt{P})0$, $r_c^2(\mathtt{C})1$, $w_c^2(\mathtt{C})0$, ... , $w_p^{n-1}(\mathtt{P})v_{n-1}b$, $r_c^{n-1}(\mathtt{P})v_{n-1}b$, $r_c^{n-1}(\mathtt{C})\bar{b}$, $w_c^{n-1}(\mathtt{C})v_{n-1}b$, $w_p^n(\mathtt{P})\bar{b}$, $r_c^n(\mathtt{P})\bar{b}$, $r_c^n(\mathtt{C})b$, $w_c^n(\mathtt{C})\bar{b}$.

```
shared var:
P: array[0..n-1] of item (initialized to (⊥ || 0))
C: array[0..n-1] of item (initialized to (⊥ || 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
   it_p : item
   bit_p : bit

in ← 0
repeat
   ... produce it_p
   bit_p ← b(P[in]) (1)
   while b(C[in]) ≠ bit_p do nothing (2)
   P[in] ← v(it_p) || ¬bit_p (3)
   in ← in + 1 mod n
until false



consumer:
var
   out : 0..n-1
   it_c : item
   bit_c : bit

out ← 0
repeat
   bit_c ← b(C[out]) (4)
   while b(it_c ← P[out]) = bit_c do nothing (5)
   C[out] ← v(it_c) || ¬bit_c (6)
   ... consume it_c
   out ← out + 1 mod n
until false
```

Figure 1: $\mathcal{ALG}_{\mathcal{S}}$, a single-writer P_1C_1-queue algorithm

Notice that for some i $(1 \leq i \leq n)$, $w_p^i(\mathtt{P})b$ could occur in $<_{L_c}$ anywhere between $r_c^{i-1}(\mathtt{P})\bar{b}$ and $r_c^i(\mathtt{P})b$. Similarly, $r_c^i(\mathtt{P})b$ could occur in $<_{L_c}$ anywhere between $r_c^{i-1}(\mathtt{P})\bar{b}$ and $w_c^i(\mathtt{C})b$ However, the read by c of P in the ith iteration must follow the write by p of P in ith iteration ensuring that the writes by p and c in the ith iteration write the same value.

By examining $<_{L_p}$ in a similar manner, it is easy to conclude that each of p and c linearizes all their writes such that these writes are interleaved. With the fact that p's writes are acts of production and c's writes are signals of consumption, the above ensures that Safety, Progress, and Order are satisfied. ∎

Corollary 4.3 *$\mathcal{ALG}_\mathcal{S}$ solves P_1C_1-queue in each of the systems of Claim 3.2 or each of the systems of Corollary 3.4.*

Proof: Each one of those systems is strictly stronger than WO. Furthermore, $\mathcal{ALG}_\mathcal{S}$ uses only single-writer objects. Therefore, the corollary follows from Claim 4.2. ∎

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. In fact, the multi-writer algorithm in Figure 2, $\mathcal{ALG}_\mathcal{M}$, does not use the trick of synchronization bits used in $\mathcal{ALG}_\mathcal{S}$ and is easier to prove correct [15]. We should also note that there is a simpler single-writer algorithm than $\mathcal{ALG}_\mathcal{S}$ that solves P_1C_1-queue in a TSO system without swap atomic instructions [15].

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, SC and PC-G (with multi-writers) systems can solve the P_mC_n-queue problem.

Claim 4.4 *There is no solution for P_1C_n-queue or P_mC_1-queue ,even for $n = 2$ or $m = 2$, in any of the systems of Claim 3.2.*

Proof: The proof is similar to that of Claim 3.2. We give the proof for P_1C_n-queue, the P_mC_1-queue case is very similar.

Suppose there is a solution to P_1C_n-queue for some system, with producer p and two consumers c and d. Consider Computation 5 where p places item ι in the queue and quits, then c removes item ι while d is idle:

$$\textbf{Computation 5} \begin{cases} p : o_1^p, o_2^p, ..., o_k^p & (p \textit{ has produced item } \iota.) \\ c : o_1^c, o_2^c, ..., o_l^c & (c \textit{ has consumed item } \iota.) \\ d : \lambda \end{cases}$$

By Progress, this computation must be possible in the system. Similarly the following computation is also possible in the system. Note that Order guarantees that c will consume item ι in Computation 5, and that d will consume the same item ι in Computation 6

$$\textbf{Computation 6} \begin{cases} p : o_1^p, o_2^p, ..., o_k^p & (p \textit{ has produced item } \iota.) \\ c : \lambda \\ d : o_1^d, o_2^d, ..., o_j^d & (d \textit{ has consumed item } \iota.) \end{cases}$$

Notice that the sequence for p is identical in both Computation 5 and Computation 6, and that p completes this sequence before c or d begin.

For each of the systems listed above, we will show that if Computation 5 and Computation 6 are possible then so is the computation:

$$\textbf{Computation 7} \begin{cases} p : o_1^p, o_2^p, ..., o_k^p & (p \textit{ has produced item } \iota.) \\ c : o_1^c, o_2^c, ..., o_l^c & (c \textit{ has consumed item } \iota.) \\ d : o_1^d, o_2^d, ..., o_j^d & (d \textit{ has consumed item } \iota.) \end{cases}$$

However, in Computation 7 both c and d have consumed the same item, contradicting the Safety requirement. Thus we can conclude that a solution to P_1C_n-queue is impossible in that system.

```
shared var: Q: array[0..n-1] of item (initialized to ⊥)


producer:
var
   in :  0..n-1
   it_p :  item

in ← 0
repeat
   while Q[in] ≠ ⊥ do nothing
   ...  produce it_p
   Q[in] ← it_p
   in ← in + 1 mod n
until false



consumer:
var
   out :  0..n-1
   it_c :  item

out ← 0
repeat
   while Q[out] = ⊥ do nothing
   it_c ← Q[out]
   Q[out] ← ⊥
   out ← out + 1 mod n
   ...  consume it_c
until false
```

Figure 2: $\mathcal{ALG}_{\mathcal{M}}$, a multi-writer P_1C_1-queue algorithm

1. Let $o_1^p, ..., o_k^p$, $(o_1^c, ..., o_l^c,\ o_1^d, ..., o_l^d)|_w$ be a total order of $(O|p \cup O_w)$ and let $(o_1^p, ..., o_k^p)|_w$, $o_1^c, ..., o_l^c$, $(o_1^d, ..., o_l^d)|_w$ be a total order of $(O|c \cup O_w)$, and let $(o_1^p, ..., o_k^p)|_w$, $o_1^d, ..., o_l^d$, $(o_1^c, ..., o_l^c)|_w$ be a total order of $(O|d \cup O_w)$. Clearly each preserves $\xrightarrow{prog}$ and satisfies condition 2 of the definition of P-RAM. Also, each is a linearization because each processor behaves exactly as it did in Computation 5 (or Computation 6) where it acted alone and is followed by a segment containing only writes. Therefore, Computation 7 is P-RAM. However, by Lemma 3.1, Computation 7 is also PC-G if it uses only single-writer objects.

2. As shown in item 1, Computation 7 is P-RAM. To prove that Computation 7 is coherent, let o_m^c be the first write to object x by c. Then, $(o_1^p, ..., o_k^p, o_1^c, ..., o_{m-1}^c, o_1^d, ..., o_j^d, o_m^c, ..., o_l^c)|_x$ is a total order on $O|x$ that clearly preserves $\xrightarrow{prog}$. Each of the segments $(o_1^p, ..., o_k^p, o_1^c, ..., o_{m-1}^c)|_x$, and $(o_m^c, ..., o_l^c)|_x$ and $(o_1^d, ..., o_j^d)|_x$ occur in a possible computation; the first two are concatenated before there has been any write to x by c so that the computation of d on x is unchanged from Computation 6. The last segment begins with a write to x which obliterate any previous change to x by d so that the computation of c on x is unchanged from Computation 5. Therefore the sequence is a linearization of $O|x$, showing that Computation 7 is coherent.

3. To see that Computation 7 satisfies TSO, we imagine a situation where p completes its computation and its write-buffer is emptied first, then both c and d consume item ι before the contents of their write-buffers are committed to main memory. Specifically, let $(O_w, \xrightarrow{writes})$ be $(o_1^p, ..., o_k^p,\ o_1^c, ..., o_l^c,\ o_1^d, ..., o_j^d)|_w$ which clearly preserves $\xrightarrow{prog}$. Let $\xrightarrow{merge_p}$ be $o_1^p, ..., o_k^p$, $(o_1^p, ..., o_k^p,\ o_1^c, ..., o_l^c, o_1^d,\ ..., o_j^d)|_w$, and $\xrightarrow{merge_c}$ be $(o_1^p, ..., o_k^p)|_w, o_1^c, ..., o_l^c, (o_1^c, ..., o_l^c, o_1^d, ..., o_j^d)|_w$, and $\xrightarrow{merge_d}$ be $(o_1^p, ..., o_k^p)|_w, o_1^d, ..., o_j^d, (o_1^c, ..., o_l^c, o_1^d, ..., o_j^d)|_w$. conditions 1, 2, 3, and 5 of TSO are obviously satisfied. Condition 4 follows by an argument similar to that in Claim 3.2 (3).

■

Corollary 4.5 *There is no solution for P_1C_n-queue or P_mC_1-queue ,even for $n = 2$ or $m = 2$, in any of the systems of Corollary 3.4.*

Corollary 4.6 *There is no algorithm that solves P_mC_n-queue in any of the systems of Claim 3.2 or any of the systems in Corollary 3.4 for $m + n \geq 3$.*

4.2 Set Problem

Although the P_mC_n-queue can not be solved in many weak systems, we show in this section that the P_mC_n-set can be solved in many of those systems.

Claim 4.7 *There is an algorithm that solves P_mC_n-set in a system that is only WO.*

Proof: Since we can solve P_1C_1-queue in such a system, P_1C_n-set can be solved by associating a separate queue with each consumer. The producer inserts its items into these queues using any discipline, say round-robin. *Safety* and *Progress* follow from the correctness of the P_1C_1-queue solution. Similarly, P_mC_1-set can be solved by associating queues to producers. To solve the general P_mC_n-set we combine these two solutions. If $m \geq n$ let consumer c_i consume from producers p_{ik} for $ik \leq n$, and similarly for $n \geq m$. ■

Corollary 4.8 *There is an algorithm that solves P_mC_n-set in each of the systems of Claim 3.2 or each of the systems of Corollary 3.4.*

Proof: Each of these systems is strictly stronger than WO. ■

Again note that the impossibility results made no assumptions about the type of the objects. So, these apply to both multi- and single-writers. Furthermore, the possibility results are proved for single-writers. Therefore, they extend to multi-writer objects. A tabulated summary of these results is given in the next section.

5 Conclusion

Sequential consistency has been frequently assumed for work in analysis of distributed algorithms. Because it is simpler and more constrained than weaker systems, it is easier to design algorithms for sequentially consistent systems. However, current and proposed multiprocessor machines implement weaker models than sequential consistency in order to enhance performance. In this paper, we have revisited the critical section problem and the producer/consumer problem in the context of memory systems that are not sequentially consistent.

Table 5 summarizes our impossibility and possibility results. We have shown that several weak memory systems cannot support a solution to the critical section problem without additional synchronization or powerful instructions. These systems include coherence, P-RAM, TSO, and PSO systems. In spite of this, we have shown that certain versions of the producer/consumer problem can be solved without any additional synchronization. In particular, even the weakest memory model, WO, can support a solution to the producer/consumer mer queue problem for two processes (a single producer and a single consumer) without the use of powerful synchronization primitives. Moreover, they can support a solution for any number of processes if the requirement of ordered consumption is removed.

Weak system architectures provide additional powerful synchronization instructions. Because these instructions are expensive, it is useful to identify ways to avoid using these primitives and incurring the corresponding performance degradation.

Table 1: Summary of possibilities ($\surd$) and impossibilities ($\otimes$)

	CSP	P_1C_1-queue	P_1C_n-queue	P_mC_1-queue	P_mC_n-queue	P_mC_n-set
TSO or PSO with basic objects	$\otimes$	$\surd$	$\otimes$	$\otimes$	$\otimes$	$\surd$
P-RAM & coherent but not PC-G	$\otimes$	$\surd$	$\otimes$	$\otimes$	$\otimes$	$\surd$
P-RAM-A	$\otimes$	$\surd$	$\otimes$	$\otimes$	$\otimes$	$\surd$
PC-G with only single-writers	$\otimes$	$\surd$	$\otimes$	$\otimes$	$\otimes$	$\surd$
PC-G with multi-writers [4]	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
$\mathrm{WO}_{coherent}$ with ordinary actions	$\otimes$	$\surd$	$\otimes$	$\otimes$	$\otimes$	$\surd$
WO with ordinary actions	$\otimes$	$\surd$	$\otimes$	$\otimes$	$\otimes$	$\surd$
SC	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

References

[1] S. V. Adve. Using information from the programmer to implement system optimizations without violating sequential consistency. Technical Report ECE 9603, Department of Electrical and Computer Engineering, Rice University, March 1996.

[2] S. V. Adve and M. D. Hill. Implementing sequential consistency in cache-based systems. *1990 Int'l Conf. on Parallel Processing*, pages I47–I50, August 1990.

[3] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):613–624, 1993.

[4] M. Ahamad, R. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 251–260, June 1993. Also available as College of Computing, Georgia Institute of Technology technical report GIT-CC-92/34.

[5] H. Attiya, S. Chaudhuri, R. Friedman, and J. Welch. Shared memory consistency conditions for non-sequential execution: Definitions and programming strategies. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 241–250, 1993.

[6] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965. Reprinted in [8].

[7] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. *Proc. of the 13th Annual Int'l Symp. on Computer Architecture*, pages 434–442, June 1986.

[8] F. Genuys, editor. *Programming Languages.* Academic Press, 1968.

[9] K. Gharachorloo, S. Adve, A. Gupta, J. Hennessy, and M. Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, August 1992.

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

[11] P. B. Gibbons, M. Merritt, and K. Gharachorloo. Proving sequential consistency of high-performance shared memories. *Symp. on Parallel Algorithms and Architectures*, pages 292–303, July 1991.

[12] J. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, March 1989.

[13] L. Higham and J. 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.

[14] L. Higham, J. Kawash, and N. 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] J. Kawash. Process coordination issues in systems with weak memory consistency. Ph.D. Dissertation draft, Department of Computer Science, The University of Calgary.

[16] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Communication of the ACM*, 17(8):453–455, August 1974.

[17] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.

[18] L. Lamport. The mutual exclusion problem. *Journal of the ACM*, 33(2):327–348, April 1986.

[19] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. on Computers*, 46(7):779–782, July 1997.

[20] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report 180-88, Department of Computer Science, Princeton University, September 1988.

[21] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[22] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[23] M. Raynal. *Algorithms for Mutual Exclusion.* The MIT Press, 1986.

[24] D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual version 9.* Prentice-Hall, 1994.