Tight Bounds for Critical Sections in Processor Consistent Platforms

Lisa Higham, Member, IEEE, and Jalal Kawash

Abstract—Most weak memory consistency models are incapable of supporting a solution to mutual exclusion using only read and write operations to shared variables. Processor Consistency—Goodman’s version (PC-G) is an exception. Ahamad et al. showed that Peterson’s mutual exclusion algorithm is correct for PC-G, but Lamport’s bakery algorithm is not. This paper derives a lower bound on the number of and type of (single or multiwriter) variables that a mutual exclusion algorithm must use in order to be correct for PC-G. Specifically, any such solution for \( n \) processes must use at least one multiwriter variable and \( n \) single-writer variables. Peterson’s algorithm for two processes uses one multiwriter and two single-writer variables, and therefore establishes that this bound is tight for two processes. This paper presents a new \( n \)-process algorithm for mutual exclusion that is correct for PC-G and achieves the bound for any \( n \). While Peterson’s algorithm is fair, this extension to arbitrary \( n \) is not fair. Six known algorithms that use the same number and type of variables are shown to fail to guarantee mutual exclusion when the memory consistency model is only PC-G, as opposed to the Sequential Consistency model for which they were designed. A corollary of our investigation is that, in contrast to Sequential Consistency, multiwriter variables cannot be implemented from single-writer variables in a PC-G system.

Index Terms—Memory consistency models, mutual exclusion, processor consistency, multiwriter/single-writer variables.

1 INTRODUCTION

The critical section problem is the most famous and well-studied problem in concurrency.\(^1\) A solution permits a set of processes to share a resource, while ensuring that no two processes access the resource concurrently. Critical section solutions for memories that satisfy Sequential Consistency\(^2\) have been known since the 1960’s; Raynal\(^3\) provides an extensive survey. In fact, as shown by Lamport\(^4\), even single-reader single-writer safe bits suffice to solve the critical section problem.

Most weak memory consistency models, however, are incapable of supporting a solution to the critical section problem using only read and write operations on shared variables\(^5\). Mutual exclusion on weak memory consistency models such as Java, Coherence, Pipelined-RAM, Total and Partial Store Ordering, Causal Memory, and several variants of Processor Consistency requires the use of expensive built-in synchronization primitives such as locks and memory barriers\(^6\). A notable exception is PC-G, which is a version of Processor Consistency\(^7\) proposed by Goodman and formalized by Ahamad et al.\(^8\). Though

1. Following Silberschatz et al.\(^9\), we refer to this problem as the critical section problem to distinguish the problem from the mutual exclusion property.

2. Several variants of Processor Consistency exist\(^10\). PC-G is due to Ahamad et al.’s\(^1\) interpretation of Goodman’s original work\(^6\).

3. The Dash and its successors Flash and Splash, originally produced by Digital Equipment Corporation, all implemented Processor Consistency models\(^15\),\(^11\).

4. L. Higham is with the Department of Computer Science, The University of Calgary, 2500 University Dr. NW, Calgary, Alberta, Canada, T2N 1N4. E-mail: higham@cpsc.ucalgary.ca.

5. J. Kawash is with the Department of Computer Science, American University of Sharjah, PO Box 26666, Sharjah, United Arab Emirates. E-mail: jkawash@ausharjah.edu.


7. For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0099-0603.

8. 1045-9219/06/$20.00 © 2006 IEEE

9. weaker than Sequential Consistency, PC-G guarantees that processes have just enough agreement about the current state of shared memory to support a solution using only reads and writes of shared variables. From a theoretical perspective, PC-G is interesting because it simultaneously guarantees two memory consistency conditions, Coherence and Pipelined RAM, that individually are very weak, yet when carefully combined provide a model powerful enough to support coordination. From a practical perspective, PC-G is important because it is one variant of the collection of Processor Consistency models that are actually provided by several machines.

Ahamad et al. have shown that Peterson’s mutual exclusion algorithm\(^17\) is correct for PC-G, but that Lamport’s bakery algorithm\(^12\) fails for PC-G\(^1\). We are thus motivated to determine what is necessary and sufficient to solve the critical section problem with only PC-G memory using only reads and writes to shared variables. For example, Peterson’s algorithm makes use of multiwriter variables, which can be written by more than one process, while Lamport’s bakery algorithm\(^12\) uses only single-writer variables, which can be written by exactly one designated process. Are multiwriter variables essential?

This paper establishes tight bounds on the number and type (single or multiwriter) of variables that a mutual exclusion algorithm must use in order to be correct for PC-G. Specifically, any PC-G solution for \( n \) processes must use at least one multiwriter variable and \( n \) single-writer variables. Furthermore, if a solution uses exactly one multiwriter and \( n \) single-writer variables, the multiwriter variable must be writable by all processes. We introduce and prove correct for PC-G a new unfair mutual exclusion
algorithm that matches this variable usage. Thus, our bound is tight for unfair solutions to the critical section problem. Since Peterson's 2-process algorithm is fair and correct for PC-G, our bound is tight even for fair solutions for two processes.

We further investigate properties that a solution, using one multiwriter variable and \( n \) single-writer variables, must satisfy in order to be correct for PC-G. Using these properties, we establish that five algorithms \cite{18}, Dekker's, Dijkstra's, Knuth's, De Bruijn's, and Eisenberg and McGuire's, do not guarantee mutual exclusion under only PC-G memory consistency. Burns' algorithm \cite{4} satisfies mutual exclusion in PC-G, but may deadlock. All of these algorithms have been developed for Sequential Consistency, and all use one multiwriter and \( n \) single-writer variables. Most of these algorithms are fair solutions for the critical section problem assuming Sequential Consistency. The only fair solution we have found for PC-G is Peterson's, which uses \( n - 1 \) multiwriter and \( n \) single-writer variables.

Since multiwriter variables are required to solve the critical section problem for PC-G, a corollary of our investigation is that, in contrast to Sequential Consistency, multiwriter variables cannot be implemented from single-writer variables with only PC-G memory consistency.

**Paper roadmap:** Section 2 provides our framework for modeling distributed systems that operate under various memory consistency constraints. It uses this framework to define the systems needed for this paper, and sets up the machinery to compare an implementation with its specification. Lower bounds are provided in Section 3 by first developing a template for our proofs (Section 3.1) and then using the template to bound the type (Section 3.2) and number (Section 3.3) of variables and to establish some necessary properties of correct PC-G algorithms (Section 3.4). Section 4 gives a new critical section algorithm that is correct under only PC-G consistency, establishing that the bounds of Section 3 are tight. Section 5 discusses the consequences of a slightly weakened definition of processor consistency. It also poses open questions related to multiwriter variables that are writeable by only a subset of the processes.

## 2 Multiprocess Systems and Memory Consistency Models

### 2.1 A Framework for Describing Systems

**Intuition:** Unlike many operational models of multiprocess computations, we do not describe a multiprocess system computation as a single sequence of events. Rather, informally, we think of each process as producing a sequence of operation invocations as dictated by its code. Any sequence of operation invocations can be turned into a sequence of operations by arbitrarily completing with a response each operation invocation that requires a response, while maintaining the original order of the invocations. An (arbitrary) system computation is formed by changing each process's sequence of operation invocations into a sequence of operations in this way. That is, a system computation is a set of sequences of operations—one for each process. A system computation satisfies a given memory consistency model if and only if it satisfies a collection of constraints expressed as partial orders on the operations of the computation. Otherwise, it is still a computation—just not one that could occur under the memory consistency model that we are considering. There are several advantages to this nonoperational perspective of computations. It allows us to capture in a natural way what each process "sees" in a computation under a given memory consistency model; it allows us to maintain the natural notion of validity for sequences of operations on objects; it provides a uniform way to describe memory consistency models; it provides a natural way to capture correctness of an implementation of one system with another. The following definitions make this intuition precise.

**Formal model:** We model a multiprocess system as a collection of processes operating on a collection of shared data objects under some partial order constraints called a memory consistency model, each of which is now defined.

**Data objects:** A data object is defined by providing its initial state, the operations that can be applied to it in each state and the change of state and/or response that results from each applicable operation. An operation has the form \( \text{act}(\text{obj}, \text{in}) = \text{out} \) where "act" is an action with input parameter "in" applied to object "obj" and that returns the output value "out." An operation \( \text{act}(\text{obj}, \text{in}) = \text{out} \) has two components; its invocation is \( \text{act}(\text{obj}, \text{in}) \) and its response is \( \text{out} \). Thus, a sequence of operation invocations on an object gives rise to a corresponding sequence of operations on it. The set of all such allowable sequence of operations associated with an object \( X \) is the specification of \( X \).

An arbitrary sequence of operations applied to object \( X \) is valid for \( X \) if and only if it is in the specification of \( X \). An arbitrary sequence \( S \) of operations (applied to possibly several objects) is valid if and only if, for each object \( X \), the subsequence of \( S \) consisting of exactly those operations applied to \( X \) is valid for \( X \). Data objects that are accessible by only one process are called local, and data objects that can be accessed by more than one process are shared.

In this paper, if the \( \text{in} \) or \( \text{out} \) parameter does not exist, it is omitted.

**Processes and multiprocesses:** An individual process is sequential computer code consisting of control structures, operation invocations on local (nonshared) objects, local computation, and operation invocations on shared objects (nonlocal invocations). A collection of individual processes is a multiprocess. Let \( J \) be a set of shared objects and let \( P \) be a multiprocess whose nonlocal operation invocations are applied to members of \( J \). Then, the pair \((P, J)\) is called a multiprocess program and \( P \) is compatible with \( J \).

To highlight the association of an operation and its invoking process, we often subscript the action by the process id, such as \( \text{act}_p(\text{obj}, \text{in}) = \text{out} \).

An individual computation of a process \( p \) is a sequence of operations, such that the order of the operations in this sequence is the same as the order in which the corresponding operation invocations appear in \( p \). This order is called
program order. A computation of a multiprocess program \((P, J)\) is a collection of individual computations, one for each process, \(p \in P\).

**Memory consistency models:** Notice that the response components of the operations in a computation are unconstrained by the preceding definition. In practice, the possible responses are determined by the architecture of the multiprocessor machine being modeled. We capture this by specifying a collection of constraints on the computations. That is, a memory consistency model is defined to be a set of partial order constraints on the operations of a computation.

**Multiprocess systems:** Let \(J\) be a set of objects and \(P\) a multiprocess compatible with \(J\), and let \(M\) be a memory consistency model. Then, the triple \((P, J, M)\) is called a multiprocess system and the couple \((J, M)\) is called a platform. The computations of a multiprocess system \((P, J, M)\) are all the computations of the program \((P, J)\) that satisfy all the constraints of the memory consistency model \(M\).

### 2.2 System Transformation and Implementation

The main tool used to establish the results of the paper is the implementation of one multiprocess system by another, which we now define. First, one multiprocess program, called the specified multiprocess program (with a corresponding specified multiprocess and objects) is transformed into another multiprocess program that uses a different set of shared objects, called the target objects, as follows. An operation on a specified object is transformed to the target objects by providing a subroutine for the operation’s invocation where the only nonlocal operation invocations of this subroutine are applied to the target objects. If the specified operation invocation produces a response, then the corresponding subroutine must return a value of the same type as this response. An object is transformed to the target object(s) by transforming each of its operations. A transformation of each shared object of a specified multiprocess program to the target objects has a natural extension to a multiprocess transformation—each operation invocation in the specified multiprocess is replaced by the subroutine for that operation invocation. The transformed multiprocess together with the target objects comprise the transformed multiprocess program.

Fig. 1 illustrates two ways of identifying a set of computations with a specified multiprocess program. Recall that if a memory consistency model is associated with the specified multiprocess program, that resulting system, by definition, has a collection of computations identified with it.

Alternatively, given a target memory consistency model, we can identify a set of computations for the specified multiprocess program via the transformation to a target platform. The transformation of a multiprocess program produces a new multiprocess system, called the transformed system, consisting of the transformed processes, the target objects and the target memory consistency model. Again, by definition, the transformed system has a set of computations identified with it—those computations of the transformed multiprocess program that satisfy the target memory consistency model. Any such computation can be interpreted as a computation of the specified multiprocess program as follows: The specified multiprocess program is just a collection of sequences of operation invocations, one for each process. By attaching to each operation invocation of this specified multiprocess program the value returned (if any) by the corresponding subroutine, we get a computation of the specified multiprocess program.

A transformation will be said to implement the multiprocess system if, informally, in Fig. 1, the set \(C'\) of computations produced by traveling the long way around is a nonempty subset of the computations \(C\) of the specified system. More precisely, let \(C\) be the set of computations of a specified multiprocess system \((P, J, M)\). Let \(\tau\) be a transformation from the objects in \(J\) to the objects in \(\hat{J}\) and denote by \(\tau(P)\) the multiprocess constructed from the extension of \(\tau\) to apply to multiprocess \(P\). Let \(\hat{D}\) be any computation of system \((\tau(P), \hat{J}, \hat{M})\) and let \(D'\) be the interpretation of \(\hat{D}\) for \((P, J)\). Then, the transformation \(\tau\) implements the system \((P, J, M)\) on the platform \((\hat{J}, \hat{M})\) if \(D' \in C\), for any such \(D'\).

### 2.3 The Critical Section Multiprocess System

In this paper, we are concerned with the Critical Section System and whether or not it can be implemented on the PC-G Platform, which has a weak memory consistency model. To address this we begin by using the framework just presented to define the systems involved.

A solution to the critical section problem is informally described as a collection of two or more processes where each one repeatedly executes in a loop the operations:

\[
\text{(remainder); (enter); (critical section); (exit);}
\]

in such a way that the following two properties are guaranteed.\(^6\) Mutual exclusion: At any time there is at most one process executing (critical section). Progress: (also called Liveness) If at least one process \(p\) is executing (enter), then eventually some process will execute (critical section). The critical section problem typically has a fairness requirement as well.\(^7\)

There is, however, no notion of global time in weak memory consistency systems. So, to investigate the possibility of implementing a solution to the critical section problem on such systems, we require a definition that is independent of time. Therefore, in what follows, the critical section problem is formalized as a multiprocess system that

---

6. Other definitions of solution properties are possible as is given by Attiya et al. [2].

7. Our impossibility and lower bound results apply, however, even to unfair solutions of the critical section problem, and therefore we will make no fairness requirement in our definition.
guarantees that the outcome of any computation is indistinguishable from one in which each process executes its (critical section) operation as if it were executing alone.

2.3.1 Time Independent Definition of a Critical Section System

Objects: A Critical Section Object $X$ supports two operation types, $\langle \text{enter}(X) \rangle$ and $\langle \text{exit}(X) \rangle$.

Any valid sequence of these operations must satisfy the following condition:

- the first operation is $\langle \text{enter}_p(X) \rangle$ for some process $p$,
- the operation immediately following $\langle \text{enter}_q(X) \rangle$ is $\langle \text{exit}_q(X) \rangle$, for any $q$, and
- if any operation follows $\langle \text{exit}_p(X) \rangle$ for any $p$, then it is $\langle \text{enter}_r(X) \rangle$ for some process $r$.

Processes: A Critical Section Process is any process $p$ with the structure:

- repeat
  - $\langle \text{remainder}_p(\cdot) \rangle$
  - $\langle \text{enter}_p(X) \rangle$
  - $\langle \text{critical section}_p(\cdot) \rangle$
  - $\langle \text{exit}_p(X) \rangle$
- until done

where $X$ is a Critical Section Object and the operations of $\langle \text{remainder}_p(\cdot) \rangle$ and $\langle \text{critical section}_p(\cdot) \rangle$ are applied to objects other than $X$.

Memory consistency model: Algorithm designers typically assume that the outcome of any computation of a multiprocess system is indistinguishable from some interleaving of the sequence of operations performed by the individual processes. This is the Sequentially Consistent memory formalized by Lamport [13]. A conceptual view of Sequential Consistency is given in Fig. 2. Let $P$ be a collection of processes operating on the collection of objects $J$. A computation $C$ of the program $(P, J)$ satisfies Sequential Consistency if there is a total order on all the operations of $C$ that extends program order and is valid for all objects in $J$.

Critical Section System: A Critical Section System is a collection of Critical Section Processes operating on a Critical Section Object $X$ and an arbitrary collection of additional objects $Y$ such that every computation is Sequentially Consistent. The system must also satisfy the progress property, which requires that if some process, $p$, is implementing $\langle \text{enter}_p(X) \rangle$, eventually some process $q$ will be implementing $\langle \text{critical section}_p(\cdot) \rangle$ under the assumption that $\langle \text{critical section} \rangle$ always eventually completes. Notice that the definitions of Critical Section Objects, Critical Section Processes, and Sequential Consistency ensure the mutual exclusion property of a Critical Section System. Specifically, because any possible computation of the processes is required to be Sequentially Consistent, there must be a single sequence of all the operations of all the Critical Section Processes that preserves program order and is valid. The validity of the Critical Section Object guarantees that each $\langle \text{enter}_p(X) \rangle$ has a subsequent $\langle \text{exit}_p(X) \rangle$ before another $\langle \text{enter}_q(X) \rangle$.

2.4 Three Weak Consistency Multiprocess Systems

We are concerned primarily with PC-G Multiprocess Systems. The PC-G Memory Consistency Model captures the combined constraints of two more basic memory consistency models. In this section, we motivate each of these two basic systems by describing the interconnection network of a simple machine. Each system is formally defined using the framework of Section 2.1. For all three systems, the processes and objects are the same, but the memory consistency model, which captures the way the processes interact with these shared objects, differ. We therefore define separately only the memory consistency model for each system. The objects and processes are defined as follows.

Objects: The only objects are read/write variables, each of which supports read and write operations. A write$_p(x, \nu)$ operation assigns a value $\nu$ to variable $x$, and a read$_p(x) = \nu$ returns to $p$ the value $\nu$ for variable $x$. A sequence of read and write operations to the variable $x$ is valid if the value returned by each read operation is the same as the value written by the most recent preceding write operation in the sequence.

Processes: A process $p$ is a read/write process if its operations on shared objects are restricted to only read and write operations on read/write variables.

2.4.1 Intuition for Coherent Systems

If a globally shared memory is partitioned into several components each of which has separate single-ported access as shown in Fig. 3, Sequential Consistency of the whole system is lost, but is maintained for each component individually. When each shared variable has its own access
channel, the memory consistency model arising from such a machine is called Coherence [5].

On this machine, program order is not necessarily maintained between operations that are applied to different variables. The machine still ensures, however, that for each shared variable individually, the execution of operations on that variable is some interleaving of the processes’ read and write operations to that variable.

2.4.2 Definition of Coherent Memory Consistency

Let \( J \) be a collection of shared read/write variables. Let \( P \) be a collection of read/write processes consistent with \( J \). Let \( O \) be all the operations of a computation \( C \) of the program \( (P, J) \). The subset of operations in \( O \) that are applied to object \( x \) is denoted \( O|x \). The program order relation on operations in \( O \) is denoted \((o, \rightarrow_p)\).

**Definition 2.1.** \( C \) satisfies Coherence if, for each variable \( x \in J \), there is a valid total order \((O|x, \rightarrow_p)\) such that

\[
(O|x, \rightarrow) \subseteq (O|x, \rightarrow_p).
\]

2.4.3 Intuition for Pipelined-RAM Systems

Consider a complete message-passing network of processes where each process stores a local copy of all of the shared memory (Fig. 4), and all of the communication channels are FIFO. Suppose reads are implemented by consulting the local memory, and writes are sent to each other process, which, upon receipt, records the write in its local copy. Then, the memory consistency model that arises is called the Pipelined-Random Access Machine (P-RAM) [16].

On this machine, processes may see the writes of other processes in differing orders. Each process, however, has a view of the execution as a sequence of all its own operations and the writes of all other processes. This sequence preserves program order and is valid.

2.4.4 Definition of Pipelined-RAM Memory Consistency

We use the same notation as is used in the definition of the Coherent Memory Consistency Model. In addition, the subset of operations in \( O \) that are by process \( p \) is denoted \( O|p \) and the subset of write operations is denoted \( O|w \).

**Definition 2.2.** \( C \) satisfies P-RAM if, for each process \( p \in P \), there is a valid total order \((O|p \cup O|w, \rightarrow_p)\) such that

\[
(O|p \cup O|w, \rightarrow_p) \subseteq (O|p \cup O|w, \rightarrow).
\]

2.4.5 Intuition for PC-G Systems

There is no natural machine that is as simple as the previous two and which motivates the definition of a PC-G System. Rather, PC-G is intended to capture those computations that could have occurred on the Coherent machine and also on the P-RAM machine.

2.4.6 Definition of PC-G Memory Consistency

We use the same notation as is used in the definitions of the Coherent and P-RAM Memory Consistency Models.

**Definition 2.3.** \( C \) satisfies PC-G if, for each process \( p \in P \), there is a valid total order \((O|p \cup O|w, \rightarrow_p)\) such that

1. \((O|p \cup O|w, \rightarrow_p) \subseteq (O|p \cup O|w, \rightarrow),\) and
2. \(\forall q \in P \) and \(\forall x \in J, (O|x, \rightarrow_p) = (O|x, \rightarrow)\).

The valid total order \((O|p \cup O|w, \rightarrow_p)\) is sometimes called the view of process \( p \).

3 Lower Bounds for Critical Sections on PC-G Platforms

This paper addresses the question of implementing the Critical Section specification on a PC-G Platform. That is, we require subroutines for the operations (enter) and (exit) whose only nonlocal operations are reads and writes to shared variables. A PC-G System is then constructed by replacing each (enter) and each (exit) operation in each process of the Critical Section System with the corresponding subroutine. This PC-G System gives rise to a set of computations—exactly those that satisfy PC-G consistency (Definition 2.3). This transformation constitutes an implementation if and only if, for any instantiation of (critical section) and (remainder), and for any such computation, the interpretation of that computation is Sequentially Consistent.

3.1 Template for Proofs

Let CSS(\(n\)) denote an \( n \)-process Critical Section System, for \( n \geq 2 \). Since an implementation must be correct for any instantiation of (critical section) and (remainder) assume that the critical section for each process, \( p \), is simply: write(name, \( p \)); read(name) where name is a read/write variable, and the remainder section is empty. By the definition of the Critical Section System, any computation arising from a collection of processes concurrently executing (enter(\(X\))); write(name, \( \cdot \)); read(name); (exit(\(X\))) has a total ordering that extends program order and is valid for both the Critical Section Object \( X \) and the read/write variable name. This implies that, for any such total order, the invocation read(name) must give the response \( p \), indicated by read(name) = \( p \), when executed by process \( p \). We consider some computations of the system when at most two processes, say \( p \) and \( q \), are participating and all remaining processes are in (remainder). If process \( p \) is executing (enter(\(X\))) while \( q \) also stays in (remainder), then by the progress property, \( p \) must subsequently execute (critical section(\(n\), \( p \))), and since the critical section is finite (two instructions) it will then execute (exit(\(X\))), producing the partial computation Computation 1, where read(name) has response \( p \).

**Computation 1**

\[
\{ p : (\text{enter}(X)); \text{write}(\text{name}, p); \text{read}(\text{name}) = p; \text{exit}(X) \}
\]

Similarly, if process \( q \) is executing (enter(\(X\))) while \( p \) stays in (remainder), Computation 2 is produced, where read(name) has response \( q \).
Computation 2
\[
\begin{align*}
p &: \{\text{remainder}(\cdot)\} \\
q &: \{\text{enter}(X); \text{write}(\text{name}, p)\}; \text{name}(\text{name}) = q; \{\text{exit}(X)\}.
\end{align*}
\]

Now, consider Computation 3, where both \( p \) and \( q \) are participating but \( \text{read}(\text{name}) \) returns \( q \) to both processes \( p \) and \( q \).

Computation 3
\[
\begin{align*}
p &: \{\text{enter}(X)\}; \text{write}(\text{name}, p)\}; \text{name}(\text{name}) = q; \{\text{exit}(X)\} \\
q &: \{\text{enter}(X)\}; \text{write}(\text{name}, q)\}; \text{name}(\text{name}) = q; \{\text{exit}(X)\}.
\end{align*}
\]

Clearly, this is not a possible computation of a Critical Section System because any possible interleaving of the operations of \( p \) and \( q \) has the value \( q \) for the valid response to \( \text{read}(\text{name}) \) by process \( p \) must have both \( p \) and \( q \) execute \( \text{enter} \) before either executes \( \text{exit} \). This violates the validity condition for the Critical Section Object \( X \). More precisely, there is no total order of the operations of Computation 3 that extends program order and is valid for both the Critical Section Object \( X \) and the read/write variable \( \text{name} \).

Our impossibility proofs proceeds as follows: Let \( P \) be a set of \( n \geq 2 \) Critical Section Processes where \( X \) is a Critical Section Object and \( \{\text{critical section}, i(\cdot)\} \) is instantiated by \( \text{write}(\text{name}, p)\}; \text{name}(\text{name}) \). Consider any transformation of the program \( \{P, \{X, \text{name}\}\} \) to a target platform with a set of read/write variables \( J \) and weak memory consistency model \( \text{M} \). Suppose the transformation of \( \{\text{enter}(X)\} \) in Computation 1 is the sequence of read/write operations \( o_1^p, o_2^p, \ldots, o_o^p \), and the transformation of \( \{\text{enter}(X)\} \) in Computation 2 is the sequence of read/write operations \( o_1^q, o_2^q, \ldots, o_o^q \). (It does not matter how \( \text{exit} \) is transformed.) Consider Computation 4, where \( q \) is returned for the value of \( \text{name} \) to both processes.

Computation 4
\[
\begin{align*}
p &: \{o_1^p, o_2^p, \ldots, o_o^p\}; \text{name}(\text{name}) = q; \{\text{exit}(X)\} \\
q &: \{o_1^q, o_2^q, \ldots, o_o^q\}; \text{name}(\text{name}) = q; \{\text{exit}(X)\}.
\end{align*}
\]

We show that Computation 4 also satisfies memory consistency model \( M \). Since the interpretation of Computation 4 is Computation 3, we have a computation of the transformed program whose interpretation is not a valid computation of a Critical Section System. This implies that the transformation is not an implementation, and we conclude that there does not exist an implementation of this Critical Section System on a target platform with only read/write variables and weak memory consistency model \( M \).

Because the \( \text{exit} \) operations in Computations 3 and 4 are irrelevant to establishing this contradiction, they will be omitted from the rest of the discussion.

None of the arguments in the following theorems depends on fairness, nor on the size of the variables, so the impossibilities apply to unfair solutions and to variables of unbounded size.

### 3.2 Bounds on Type of Variables

A \textit{k-writer} (read/write) variable can be written by up to \( k \) designated processes. Any \textit{k-writer} variable where \( k \geq 2 \) is called a \textit{multiwriterv (read/write) variable}; if \( k = 1 \), the variable is called a \textit{single-writer (read/write) variable}. This section shows that the use of multiwriter variables is crucial to implement a Critical Section System on a PC-G Platform. The argument used is that with only single-writer variables, PC-G is no stronger than P-RAM and that with only P-RAM consistency, implementing CSS(\( n \)) is impossible.

**Lemma 3.1.** Let \( J \) be a set of only single-writer variables. Then, for any set of processes \( P \) compatible with \( J \), the systems \( (P, J, \text{PC-G}) \) and \( (P, J, \text{P-RAM}) \) are equivalent.

**Proof.** Since PC-G is at least as strong as P-RAM, it suffices to show that without the use of multiwriter variables, P-RAM is at least as strong as PC-G. Let \( O \) be all the operations of any computation of the system \( (P, J, \text{P-RAM}) \). Let \( O \{\text{P} \cup O|w, L^-o\} \) and \( O \{\text{P} \cup O|w, L^-o\} \) be valid total orders for \( p \) and \( q \in P \) that are guaranteed by P-RAM consistency. Since, for any variable \( x \in J \), there is only one process, say \( s \), that writes to \( x \), and both \( O \{\text{P} \cup O|w, L^-o\} \) and \( O \{\text{P} \cup O|w, L^-o\} \) have all these writes to \( x \) in the program order of \( s \), the order of the writes to \( x \) in \( O \{\text{P} \cup O|w, L^-o\} \) is the same as the order of the writes to \( x \) in \( O \{\text{P} \cup O|w, L^-o\} \). Therefore, the definition of PC-G (Definition 2.3) is satisfied.

Theorem 3.2 has been proved by Attiya and Friedman [3] using an appealing technique. They define a memory consistency model to be \textit{fast} if there is a machine that implements it and executes each operation in time that is asymptotically faster than network delay. They then show that 1) there is no solution to the critical section problem as defined in this paper on a fast machine, and 2) P-RAM is fast. We provide an alternative proof here that is independent of machine implementations, and is a simple example of the use of our framework.

**Theorem 3.2 [3].** There is no implementation of CSS(\( n \)) on a P-RAM Platform, for any \( n \geq 2 \).

**Proof.** Apply the template of Section 3.1. Define the following sequences for \( p \) and \( q \), respectively, for Computation 4. For a sequence \( S \) of read/write operations, \( S'|w \) denotes the subsequence of \( S \) containing only the write operations.

\[
\begin{align*}
O\{\text{P} \cup O|w, \overset{\text{P-RAM}}{\rightarrow}\} &= o_1^p, \ldots, o_o^p, \text{write}(\text{name}, p), \\
&\text{(o}_1^q, \ldots, o_o^q|w, \text{write}(\text{name}, q), \\
&\text{read}(\text{name}) = q
\end{align*}
\]

\[
\begin{align*}
O\{\text{P} \cup O|w, \overset{\text{P-RAM}}{\rightarrow}\} &= o_1^q, \ldots, o_o^q, (o_1^p, \ldots, o_o^p)|w, \text{write}(\text{name}, p), \\
&\text{write}(\text{name}, q), \text{read}(\text{name}) = q
\end{align*}
\]

Clearly, each preserves program order \((\overset{\text{P-RAM}}{\text{prog}})\) as required by the definition of P-RAM. Also, each is valid because the first part (for instance, \( o_1^p, \ldots, o_o^p \)) corresponds to a possible computation, and the second part (for instance, \( (o_1^q, \ldots, o_o^q)|w \)) contains only writes, and the remaining subsequence of writes and reads to \( \text{name} \) is valid. Thus, Computation 4 satisfies P-RAM. Since its interpretation is not valid for a Critical Section System,
the transformation is not an implementation of a Critical Section System.

Theorem 3.3. There is no implementation of CSS\( (n) \) on a PC-G Platform that uses only single-writer variables, for any \( n \geq 2 \).

Proof. This follows immediately from Lemma 3.1 and Theorem 3.2.

Ahamad et al. [1] prove that Lamport’s bakery algorithm [12], which uses only single-writer variables, is incorrect for PC-G. By Theorem 3.3, any solution to the critical section problem on a PC-G Platform must use at least one multiwriter variable. Vitanyi and Awerbuch [21] showed that multiwriter variables can be implemented in a wait-free manner using single-writer variables in Sequentially Consistent Systems. In contrast, in PC-G, there is no (even nonwait-free) construction of multiwriter variables from single-writer variables.

Corollary 3.4. No multiwriter variable can be implemented from any number of single-writer variables in PC-G memory systems.

Proof. Peterson’s algorithm solves CSS\( (k) \) for PC-G systems with \( k \geq 2 \) using \( k \)-writer variables, and there is no solution with only single-writer variables by Theorem 3.3.

3.3 Bounds on Number of Variables

After showing that at least one multiwriter variable is required, a natural question is: “What is the minimum number of variables needed to implement CSS\( (n) \) on a PC-G Platform?”

Theorem 3.5. There is no implementation of CSS\( (n) \) on a PC-G Platform that uses only one multiwriter variable, and fewer than \( n \) single-writer variables for any \( n \geq 2 \).

Proof. Assume that there is a transformation that uses only one multiwriter variable (even an \( n \)-writer) and fewer than \( n \) single-writer variables and implements CSS\( (n) \) on PC-G. Since there are \( n \) processes, and fewer than \( n \) single-writer variables there is at least one process, say \( p \), that does not write to any single-writer variable. Computations 1 and 2 must exist due to the progress property. We show that Computation 4 satisfies PC-G.

Let \( \alpha_i^q \) be \( q \)'s first write to the multiwriter variable. Consider the following orders for processes \( p \) and \( q \) from Computation 4.

\[
(O[p \cup O][w, L_p]) = \alpha_0^p, \ldots, \alpha_n^p, \text{write(name, } p),
\]
\[
(\alpha_1^p, \ldots, \alpha_i^p)[w, \text{write(name, } q),
\]
\[
\text{read(name) } = q
\]

\[
(O[q \cup O][w, L_q]) = \alpha_1^q, \ldots, \alpha_{n-1}^q, (\alpha_0^q, \ldots, \alpha_i^q)[w, \text{write(name, } p),
\]
\[
\alpha_1^q, \ldots, \alpha_i^q,
\]
\[
\text{write(name, } q), \text{read(name) } = q
\]

Both sequences maintain program order. The sequence for \( p \) is valid because it consists of Computation 1 without the final read followed by writes by \( q \) and a final valid read by \( p \). The sequence for \( q \) is also valid. The segment \( \alpha_1^q, \ldots, \alpha_{n-1}^q \) does not contain any writes to the multiwriter variable. Since \( p \) does not write to any single-writer variable, the segment \( (\alpha_0^p, \ldots, \alpha_i^p) \) contains only writes to the multiwriter variables. The segment \( \alpha_0^q, \ldots, \alpha_i^q \) starts with a write to the multiwriter variable overwriting any changes the segment \( (\alpha_0^p, \ldots, \alpha_i^p) \) caused, and the final read of name by \( q \) is valid. Therefore, both are valid total orders.

Also, each total order lists \( p \)'s writes to the multiwriter variable followed by \( q \)'s. Since only \( q \) writes to any single-writer variables, the two sequences also agree on the order of this variable. So, both sequences agree on the order of writes for each variable. Hence, all conditions of Definition 2.3 are satisfied.

When \( n = 2 \), the bound of Theorem 3.5 is tight, even if all variables are allowed to be multiwriter variables.

Theorem 3.6. Two variables are insufficient to implement CSS\( (2) \) on a PC-G Platform.

Proof. Assume that there is an implementation that uses exactly 2 variables, say \( x \) and \( y \), and implements CSS\( (2) \) on a PC-G Platform. Then, Computations 1 and 2 exist. We show that Computation 4 satisfies PC-G.

Partition the computation of \( p \) in Computation 4 into subsequences \( S_0^p, S_1^p, \ldots, S_k^p \) where each subsequence \( S_i^p \) is defined by:

1. \( S_0^p \) contains all operations from \( \alpha_0^p \) up to but not including the first write by \( p \), labeled \( \alpha_0^p \).
2. \( S_i^p, i \geq 1 \), contains all operations from \( \alpha_i^p \) up to but not including the first write, labeled \( \alpha_{i-1}^p \), such that \( \alpha_0^p \) and \( \alpha_{i-1}^p \) are applied to different variables.

Similarly, partition the computation of \( q \) in Computation 4 into subsequences \( S_0^q, S_1^q, \ldots, S_k^q \).

The subsequence \( S_0^p \) is either empty or consists entirely of reads returning initial values. Each subsequence \( S_i^p \), \( i \geq 1 \) starts with a write and all the writes in \( S_i^p \) are applied to the same variable. If the writes in \( S_i^p \) are applied to \( x \), \( S_i^p \) is called \( x \)-gender; otherwise, it is called \( y \)-gender. Note that the identified subsequences of the sequences \( S_0^p, S_1^p, \ldots, S_k^p \) and \( S_0^q, S_1^q, \ldots, S_k^q \) alternate in gender.

To see that Computation 4 satisfies PC-G, consider two cases (the other two are symmetric). \( S_i^p \) is an \( x \)-gender but \( S_i^q \) is a \( y \)-gender: Define the following total orders:

\[
(O[p \cup O][w, L_p]) = S_0^p, S_1^p, (S_2^p)[w, S_2^p, \ldots, (S_i^p)[w, S_i^p, \ldots, write(name, p),
\]
\[
write(name, q), \text{read(name) } = q
\]

\[
(O[q \cup O][w, L_q]) = S_0^q, S_1^q, (S_2^q)[w, S_2^q, \ldots, (S_i^q)[w, S_i^q, \ldots, write(name, p),
\]
\[
write(name, q), \text{read(name) } = q
\]

Clearly, \( (O[p \cup O][w, L_p]) \) and \( (O[q \cup O][w, L_q]) \) maintain program order. They are also valid because, for each \( i \geq 1 \), \( S_i^p \) is the same gender as \( S_{i-1}^p \) and \( S_{i+1}^q \). Therefore, inserting \( (S_{i-1}^p)[w \) immediately before \( S_i^p \) does not affect \( p \)'s computation because \( S_i^p \) starts with a write that
Fig. 5. Solutions to the critical section problem for sequential consistency.

| Algorithm                    | Year | $|P|$ | Single-writer Variables | Multi-writer Variables | Flag Values | Fairness Delay | Correct for PC-G |
|------------------------------|------|-----|-------------------------|------------------------|-------------|----------------|------------------|
| Dekker’s                     | 1965 | $n = 2$ | $n$                    | $1$                     | $2$         | $\infty$      | NO               |
| Dijkstra’s                   | 1965 | $n \geq 2$ | $n$                    | $1$                     | $3$         | $\infty$      | NO               |
| Knuth’s                      | 1966 | $n \geq 2$ | $n$                    | $1$                     | $3$         | $2^{n-1}$     | NO               |
| De Bruijn’s                  | 1967 | $n \geq 2$ | $n$                    | $1$                     | $3$         | $(n^2 - n)/2$ | NO               |
| Eisenberg and MacGuire’s     | 1972 | $n \geq 2$ | $n$                    | $1$                     | $3$         | $n - 1$       | NO               |
| Burns’                       | 1981 | $n \geq 2$ | $n$                    | $1$                     | $2$         | $\infty$      | NO               |
| Peterson’s                   | 1981 | $n \geq 2$ | $n$                    | $n - 1$                 | $2$         | $(n^2 - n)/2$ | YES              |
| CSS$_{PC-G}$ (Section 4)     | 2003 | $n \geq 2$ | $n$                    | $1$                     | $2$         | $\infty$      | YES              |

3.4 Additional Requirements for Correct Implementations

By Theorems 3.3 and 3.5, an algorithm that correctly implements CSS($n$) on PC-G must use at least $n$ single-writer variables and one multiwriter variable. Many algorithms that implement CSS($n$) on a Sequentially Consistent Platform use exactly this number and type of variables. In particular, all the algorithms discussed in this section except Peterson’s (which uses $n$ single-writer and $n - 1$ multiwriter variables) use one $n$-writer variable, turn, written and read by every process, and $n$ single-writer variables, flag[$i$] for each $i$, written only by process $i$ and read by every process other than $i$. These algorithms are quoted in Appendix A, which can be found on the Computer Society Digital Library at http://www-computer.org/tpds/archives.htm and listed in Fig. 5, which characterizes each algorithm by four attributes: number of processes $|P| = n$, number of variables, number of values that a flag variable can be assigned, and fairness delay. This fairness delay is the maximum total number of times other processes can execute (critical section) before a designated process executing (enter) moves to executing (critical section) assuming a Sequentially Consistent memory. When there is no upper bound on the fairness delay (denoted $\infty$), the algorithm is prone to starvation, and is thus unfair.

Although this number of variables is a necessary requirement for a PC-G solution, we show next that these algorithms do not implement CSS($n$) on PC-G. First, we derive some general restrictions that all correct solutions for PC-G must satisfy. Then, one of these restrictions is used to show that Dekker’s, Dijkstra’s, Knuth’s, De Bruijn’s, and Eisenberg and MacGuire’s do not implement CSS($n$) on PC-G.

Lemma 3.8. Any implementation of CSS($n$) on a PC-G Platform that uses exactly $n$ single-writer variables and one multiwriter variable must satisfy each of the following properties for every process $p$ and Critical Section Object $X$:

1. In the implementation of $\langle$enter$_p$(X)$\rangle$, $p$ writes a single-writer variable at least once.
2. In the implementation of $\langle$enter$_p$(X)$\rangle$, $p$ writes the multiwriter variable at least once. Furthermore, this implementation includes at least one operation subsequent to some such write.
3. In the implementation of $\langle$enter$_p$(X)$\rangle$, $p$ reads every other single-writer variable.

Proof. We follow the proof template of Section 3.1:

1. Assume it is not the case; then, there is at least one process, say $p$, that does not write to any single-writer variable. The valid total orders used in Theorem 3.5 apply.
2. Assume that a process $q$ either does not write the multiwriter variable in $\langle$enter$_q$(X)$\rangle$ or does write the multiwriter variable exactly once but this write operation is the last operation of $\langle$enter$_q$(X)$\rangle$ (i.e., $o^q_n$). Under this assumption, Computation 4 satisfies PC-G as shown by the following sequences.
Proof. The following algorithms do not implement Corollary 3.10. known algorithms.

In any implementation of CSS(\(n\)) on a PC-G Platform that uses exactly \(n\) single-writer variables and one multiwriter variable, the multiwriter variable must be an \(n\)-writer.

Another corollary arises by applying Lemma 3.8 to well-known algorithms.

Corollary 3.10. The following algorithms do not implement CSS(\(n\)) on a PC-G Platform, for any \(n \geq 2\): Dijkstra’s, Dekker’s, De Bruijn’s, Knuth’s, Eisenberg, and MacGuire’s algorithms.

Proof. In Dijkstra’s algorithm, if the multiwriter variable \( \text{turn} \) is initially \(p\), \(p\) can progress to (critical section) without writing to the multiwriter variable. In Dekker’s and Bruijn’s algorithms, the multiwriter variable is only written in (exit). In Knuth’s, and in Eisenberg and MacGuire’s algorithms, the multiwriter variable is only written as the last operation of (enter). By Lemma 3.8, item 2, none of these algorithms represents an implementation of CSS(\(n\)) on a PC-G Platform.

Burns’ algorithm: Burns’ algorithm [4] is given in Appendix A, which can be found on the Computer Society Digital Library at http://www.computer.org/tpds/archives.htm. In an earlier version of this work [9], we showed that Burns’ algorithm satisfies mutual exclusion for PC-G Platforms. We also showed that Burns’ algorithms satisfies progress for PC-G Platforms under some additional constraints. Our proof implicitly assumed that the PC-G implementation guarantees a form of causality ensuring that if a process \(p\) reads a value written by another process \(q\), all the operations that preceded this write in \(q\)’s program must have been directly (writes) or indirectly (reads) observed in \(p\)’s view before its read. The definition of PC-G, however, does not give this guarantee; without it, Burns’ algorithm can deadlock (even for two processes) as confirmed by Computation 5. The segments enclosed in square brackets represent one iteration and each is being repeated indefinitely.

Computation 5

\[
\begin{align*}
\{p\} & \colon [\text{write}(\text{flag}[p], \text{true}), \text{write}(\text{turn}, p), \text{read}(\text{turn}) = q, \\
& \quad \text{write}(\text{flag}[q], \text{true}), \text{write}(\text{flag}[q], \text{false}), \text{read}(\text{flag}[q]) = \text{false}] , \ldots \\
\{q\} & \colon [\text{write}(\text{flag}[q], \text{true}), \text{write}(\text{turn}, q), \text{read}(\text{turn}) = p, \\
& \quad \text{write}(\text{flag}[q], \text{false}), \text{read}(\text{flag}[p]) = \text{false}] , \ldots
\end{align*}
\]

The total orders given in Fig. 6 show that Computation 5 satisfies PC-G. It can be easily confirmed that each is valid and maintains program order. Furthermore, the writes to \(\text{turn}\) in both sequences appear in the same order.

4 Critical Section Algorithms for \(n\)-Process PC-G Platforms

Ahmad et al. [1] proved that Peterson’s algorithm [17], which was originally developed for Sequentially Consistent Systems, implements CSS(2) for PC-G Platforms. The well-known tournament tree structure can be used to extend Peterson’s algorithm to implement CSS(\(n\)) on PC-G Platforms.
as explained in Appendix B, which can be found on the Computer Society Digital Library at http://www.computer.org/tpds/archives.htm. This tournament tree implementation uses $2n - 1$ variables—as many as Peterson’s $n$-process algorithm uses (Appendix A, which can be found on the Computer Society Digital Library at http://www.computer.org/tpds/archives.htm). To establish that the lower bound is tight for $n$ processes, we present a new algorithm, called CSS$_{PC-G}$, that solves CSS($n$) on a PC-G Platform, using the minimum number and type of variables: $n$ single-writers and one $n$-writer. However, CSS$_{PC-G}$ is unfair, and the bound is known to be tight only for unfair solutions.

Algorithm CSS$_{PC-G}$ is given in Fig. 7 and is compared with the seven well-known critical section algorithms in Fig. 5. In this algorithm, a process with a higher identifier has a higher priority to execute its (critical section) operation. A process with a smaller identifier can starve executing lines 2 and 3.

**Theorem 4.1.** CSS$_{PC-G}$ is an (unfair) implementation of CSS($n$) on PC-G Platforms.

**Proof:** Mutual exclusion. Assume for the sake of contradiction that there exists some PC-G computation of CSS$_{PC-G}$, where two processes, say $p$ and $q$, with $p < q$, execute in the critical section “concurrently.” That is, there is a (partial) computation of the implementation that has the interpretation:

![Algorithm CSS$_{PC-G}$ for CSS($n$) on PC-G Platforms.](image)

In Computation 7, the second set of ellipses for each process only includes reads of flags of processes other than $p$ and $q$ and can be ignored.

Since Computation 7 satisfies PC-G, there must be two valid total orders $(O|p \cup O|w, \rightarrow)$ and $(O|q \cup O|w, \rightarrow)$ that preserve program order and that agree on the order of writes to turn. Suppose write$(turn, p)$ precedes write$(turn, q)$ in both orderings. Then, for $q$,

$$\text{write}(flag[p], true) \xrightarrow{L_2} \text{write}(turn, p) \xrightarrow{L_2}$$

$$\text{write}(turn, q) \xrightarrow{L_2} \text{read}(flag[p]) = false.$$

This cannot be valid without a write$(flag[p], false)$ somewhere between the first and last operations of this subsequence. However, no such write can exist, given that write$(turn, p)$ is the last write by $p$ before it completes the implementation of $(enter_p(X))$. If, otherwise, write$(turn, q)$ precedes write$(turn, p)$ in both orderings, then a very similar argument shows that there is no way to provide the required valid total order $(O|p \cup O|w, \rightarrow)$ for Computation 7.

**Progress:** Call the while loop at lines 2 and 3 the first-while and the while loop at line 6 the second-while. Assume for the sake of contradiction that there is a PC-G computation of CSS$_{PC-G}$ with $m \geq 2$ processes, that without loss of generality have identifiers in $\{1, 2, \cdots, m\}$, executing the operation $(enter)$, but none of them finishes it.

First, note that process $m$ never executes the body of the first-while because it has the highest identifier. Hence, $m$ only writes $true$ to $flag[m]$ and eventually this will be realized in the view of each process and subsequently will never change. If a process $i \neq m$ waits in the first-while, $i$ will never be able to escape this loop since $m$ is also forever stuck in $(enter)$.

Second, process 1 never executes the body of the second-while because it has the smallest identifier. When $flag[m]$ becomes $true$ in the view of process 1, this process will eventually wait forever in the first-while. Now, process 1 only writes $false$ to $flag[1]$ and eventually this will be realized in the view of each process and subsequently will never change. Inductively, this applies to all processes in $\{2, \cdots, m - 1\}$ in order. This results in all processes, except $m$, waiting in the first-while and their flags become $false$ in all the PC-G views forever.

However, when the view of process $m$ includes $flag[i]$ is $false$, $\forall i \neq m$, the second-while condition will always be false for $m$. Process $m$ must be repeatedly executing lines 4, 5, and 7. Since process $m$ is the only process writing to turn, the condition at line 7 will eventually be satisfied in $m$’s view, and it will finish $(enter)$, making progress. $\square$

### 5 Further Comments and Future Work

PC-G is a consistency model that satisfies both P-RAM consistency and Coherence. Furthermore, for any computation, for each process $p$, there must be a single valid total order of the write operations and all operations by $p$ that
higham and kawash: tight bounds for critical sections in processor consistent platforms

satisfies both requirements simultaneously. Even the slight relaxation to a consistency model that is the intersection of both P-RAM and Coherence (but which permits distinct orderings for each requirement) is too weak to support an implementation of a Critical Section System without using stronger objects than simple variables (even unbounded ones). This can be proved with techniques similar to ones used here [10]. Thus, PC-G appears to be the weakest memory consistency model in the literature that has a solution to the critical section problem using only reads and writes to shared variables.

We proved that any implementation of CSS(n) on a PC-G Platform must use at least one multiwriter variable. Furthermore, if only one multiwriter variable is used, we need an additional n single-writer variables and the multiwriter variable must be an n-writer. A new algorithm, which uses one n-writer and n single-writer variables is proved correct for PC-G and establishes that this bound is tight. But, our new algorithm is unfair. Peterson’s algorithm for two processes, which uses one multiwriter and two single-writer variables and is correct and fair for PC-G, shows that this lower bound is tight even for fair solutions when n = 2. It is our conjecture that a fair implementation for n ≥ 3 processes cannot be constructed using only one multiwriter and n single-writer variables. Given our unfair solution that meets these bounds, any proof of this conjecture must exploit fairness.

Many other algorithms that use the same number and type of variables as our new algorithm have been shown to fail for PC-G. Finally, Peterson’s algorithm, which uses n – 1 multiwriter and n single-writer variables, is correct and fair for PC-G.

In Distributed-Shared Memory (DSM) systems, multicasting to a group of processes may be more desirable than broadcasting to all processes in the system. Since PC-G is a DSM system, this motivates the questions:

1. Is there an implementation of CSS(n) on a PC-G Platform using k-writer variables where k is the group size (and typically much smaller than n)?

2. For what values of k can an n-writer variable be constructed from k-writer variables, for PC-G systems?

In this paper, question 1 is solved only for k = n, and question 2 is answered (negatively) only for k = 1. The lower bound techniques in this paper are easily generalized to show that on a PC-G system, using k-writer variables either to construct an n-writer variable or to solve CSS(n) requires at least three k-writer variables, for k ≤ n – 1 and requires at least \( \lceil \frac{n}{k} \rceil - 1 \) 2-writer variables. Beyond these straightforward extensions, the techniques in this paper do not seem to help us address these questions for general values of k.

Our goal is to provide efficient implementations of common process coordination specifications for various systems with weak memory consistency. Since use of built-in synchronization primitives in these systems is generally expensive, the first step is to determine if solutions exist that only use read/write variables. Surprisingly, even weaker models than PC-G can support solutions to some restricted coordination problems using only read/write variables [8]. Future work will examine other common problems and memory consistency models. For those that require some use of synchronization primitives, we hope to propose algorithms that minimize the amount of synchronization and use the cheapest primitives possible.

ACKNOWLEDGMENTS

The authors thank the anonymous referees for their comments that helped them clarify their presentation. A preliminary version of this paper appeared in the Proceedings of the 14th International Symposium on Distributed Computing (DISC 2000) [9]. L. Higham was supported in part by the Natural Sciences and Engineering Research Council of Canada grant OGP041900. J. Kawash is also affiliated with the Department of Computer Science at The University of Calgary, Calgary, Canada.

REFERENCES


Lisa Higham's research interest is algorithms for distributed computation. Specific topics include fault tolerance and weak memory consistency models, arising from distributed shared memory. As associate head, she enlarged and enriched the graduate program in computer science at the University of Calgary, Canada. She has served on program committees for conferences on distributed computation and many committees that adjudicate graduate and postdoctoral candidates for fellowship support. She is a member of the IEEE and the IEEE Computer Society.

Jalal Kawash received the PhD degree from the University of Calgary in June 2000. He is currently an assistant professor at the American University of Sharjah, United Arab Emirates, and an adjunct assistant professor at the University of Calgary, Canada. His research interests are in distributed systems, memory consistency models, mobile virtual communities, and computing education.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.