Implementing Sequentially Consistent Programs on Processor Consistent Platforms

Lisa Higham[(1)] and Jalal Kawash[(1),(2)]

higham@cpsc.ucalgary.ca jkawash@ausharjah.edu

[(1)]Department of Computer Science, The University of Calgary, Calgary, Canada

[(2)]Department of Computer Science, American University of Sharjah, Sharjah, UAE

Abstract

Designers of distributed algorithms typically assume strong memory consistency guarantees, but system implementations provide weaker guarantees for better performance and scalability. This motivates the study of how to implement programs designed for sequential consistency on platforms with weaker consistency models. Typically, such implementations are impossible using only read and write operations to shared variables. For example, Sparc's Total Store Order machines (and their even weaker Partial Store Order and Relaxed Memory Order machines), the PowerPC, Alpha, Java and most variants of processor consistency all require the use of strong and expensive built in hardware synchronization primitives to implement mutual exclusion. One variant of processor consistency originally proposed by Goodman and called here PC-G is an exception. It is known that PC-G provides just enough consistency to implement mutual exclusion using only reads and writes. This paper extends the study of Goodman's version of processor consistency from specific problems (such as mutual exclusion) to arbitrary ones. That is, we investigate the existence of compilers to convert programs that use shared read/write variables with sequentially consistent memory semantics, to programs that use read/write variables with PC-G consistency semantics.

We first provide a simple program transformation, and prove that it compiles any 2-process program with only single-writer variables from sequentially consistency to PC-G consistency. We next prove that no similar simple compiler exists for even a very restricted class of 3-process programs.

Even though our program transformation is not a general compiler for 3 or more processes, it does correctly transform some specific *n*-process programs from sequentially consistency to PC-G consistency. In particular, for the special case of the (necessarily randomized) Test&Set algorithm of Tromp and Vitanyi, our transformation extends to any number of processes. Thus, one notable outcome is an implementation of Test&Set on PC-G that uses only reads and writes of shared variables. This is the first expected, wait-free implementation of Test&Set on any weak memory model, and illustrates the use of randomization with a weak memory model.

Keywords: Memory consistency models, sequential consistency, processor consistency, program transformation, expected wait-free Test&Set.

1 Introduction

Designers of distributed algorithms typically assume strong memory consistency guarantees, which ensure a high degree of agreement between the processes' views of the state of the shared objects. One of the strongest models, Lamport's sequential consistency [11], requires agreement on a single view of the objects' state between all the system processes. However, actual systems provide weaker consistency guarantees than sequential consistency in order to achieve better scalability, performance, and availability. While it is easier to design algorithms for strong consistency models, the systems with weaker consistency guarantees are more efficient. This motivates us to investigate how to implement algorithms that are designed assuming sequentially consistency on systems that have weaker consistency guarantees.

Such an implementation can be complicated. Since, for most weak models, even mutual exclusion is impossible using only reads and writes to shared variables, algorithms typically need to be augmented with hardware specific synchronization primitives to maintain correctness. For example, Sparc's Total Store Order machines (and their even weaker Partial Store Order and Relaxed Memory Order machines), the PowerPC, Alpha, Java and most variants of processor consistency all require the use of strong and expensive built in hardware synchronization primitives to implement mutual exclusion [6, 8]. Processor Consistency – Goodman's version [2, 1] (abbreviated, PC-G) is an exception[1]. Though weaker than sequential consistency, PC-G guarantees that processes have just enough agreement about the current state of shared memory to support synchronization using only reads and writes of shared variables[2]. Some sequentially consistent algorithms that use read/write operations are correct for PC-G without any modifications. For example, Ahamad *et. al.* have shown that Peterson's mutual exclusion algorithm [15] is correct for PC-G. Lamport's bakery algorithm [10], however, fails for PC-G [1]. Previously [7], in an effort to understand the impact of weakening the consistency guarantees, we studied the implementation of mutual exclusion algorithms on PC-G platforms[3]. We established tight bounds on the number and type (single- or multi-writer) of variables that a mutual exclusion algorithm must use in order to be correct for PC-G; we determined that most, but not all, of the well known mutual exclusion algorithms [16], which were developed for sequential consistency, fail under only PC-G memory consistency; and we showed that, in contrast to sequential consistency, multi-writer variables cannot be constructed from single-writer variables in a PC-G memory [7].

Previous work has not addressed the more general problem of determining necessary and sufficient conditions for implementing an arbitrary sequentially consistent algorithm on a given weak platform with just read/write variables. This paper reports progress towards this more general goal. Since PC-G is known to support at least mutual-exclusion, while most other models cannot, we begin our study with the PC-G model. Specifically, using the addition of only one multi-writer variable, we provide a simple implementation on a PC-G platform of any 2-process sequentially consistent program that uses only single-writer variables. We also prove that our transformation, and any other meeting some specific constraints, can fail to correctly implement even a restricted

[1]The term processor consistency has been used to refer to a range of similar but subtly different memory models [19]. The one referred to in this paper is due to Ahamad *et. al.*'s [1] interpretation of Goodman's original work.

[2]Even among all the processor consistency variants, only PC-G is strong enough to support a solution to mutual exclusion without strong synchronization primitives.

[3]We use *platform* to mean a collection of shared objects together with their consistency guarantees. The PC-G platform has read/write variables and PC-G consistency.

class of 3-process sequentially consistent systems on PC-G platforms (Section 3). Our transformation is therefore restricted to just 2 processes. However, there are potentially practical applications because there are 2-process algorithms that use only single-writer variables and solve synchronization problems in sequentially consistent systems. One example is Lamport's bakery algorithm for mutual exclusion [10]. We have addressed its correctness for PC-G previously [7]. Another is Tromp and Vitanyi's remarkable randomized algorithm for expected wait-free Test&Set [18]. In Section 4 we use our transformation to implement the 2-process expected wait-free Test&Set of Tromp and Vitanyi on a PC-G platform. Then we exploit additional properties of the Test&Set problem to extend it to the *n*-process case (even though this method is shown to fail even for 3-processes for some other programs.) So, as a consequence of our investigation, we achieve an implementation of Test&Set on a model weaker than sequential consistency using only read/write variables. This is the first expected wait-free Test&Set on a consistency model that is weaker than sequential consistency.

In order to make our results precise, we begin by providing a framework for modeling and comparing systems with different memory consistency models (Section 2). The framework is of independent interest because it allows us to define transformations between systems at various levels of abstraction and provides the strategy for proving their equivalence. The remainder of the paper is simplified by exploiting the tools and methods of this framework.

Our goal is to ensure that computations of our transformed algorithms are indistinguishable from sequentially consistent computations of the original algorithms. It may be asked why we do not, instead, strive to ensure the even stronger guarantee of Linearizable memory consistency, which was formalized by Herlihy and Wing [4]. Linearizability requires that the sequentially consistent total order also agrees with the real time order when operations actually occurred. This requires some notion of the real time interval of an operation, which is not available to processors in a fully asynchronous system. Since there is no notion of global time in weak memory consistency models, we restrict our correctness requirement to sequential consistency.

2 Framework

2.1 Multiprocess systems

A multiprocess system can be modeled as a collection of *processes* operating on a collection of *shared data objects* under some partial order constraints called a *memory consistency model*. To define a system, we specify each of these components.

Objects: A shared data object is defined by providing its initial state and the operation invocations that can be applied to the object, and specifying the set of allowable sequences of operations that arise from a sequence of such invocations. 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. All sections except Section 4 consider only variables that can be written and read, and which are defined as follows.

A *read/write variable*, *x*, is initially undefined, and supports the operations: $\text{write}(x, \nu)$, which assigns a value ν to variable *x*, and $\text{read}(x) = \nu$, which returns the value ν for variable *x*. A sequence of read and write operations to 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.

A read/write variable is a *single-writer variable* if it can be written by only one process, otherwise it is a *multi-writer variable*.

Each operation has an invocation and a response component. For the operation $\text{read}(x) = \nu$ the invocation component is $\text{read}(x)$ and the response is ν. For the operation $\text{write}(x, \nu)$, the invocation component is also $\text{write}(x, \nu)$ and the response is an acknowledgement that is ignored for this paper.

Processes: A *process* in a multiprocess system is just sequential code whose (non-local) operation invocations are applied to objects in the system. The order of the operation invocations by any process induces a total order (per process) on its resulting operations, and a partial order called *program order* on any collection of operations, O. This partial order is denoted $(O, \xrightarrow{prog})$.

Let P be a set of processes whose operation invocations are applied to shared objects in J. Then the pair (P, J) is called a *(multiprocess) program* and P is *consistent* with J.

This section and Section 3 are concerned with processes whose non-local operation invocations are applied to shared read/write variables.

Memory consistency models: A *computation* of a multiprocess program (P, J) is a collection of sequences of operations, one for each process $p \in P$, where the operation invocations in p's sequence arise from p's code and are in p's program order.

Notice that the response components of the operations in a computation are not constrained by the preceding definitions. Rather, the possible responses to the invocations are determined by the architecture, which we model as a set of memory consistency constraints. Thus a *memory consistency model* is a set of partial order constraints on the operations of a computation. These partial orders are usually defined on subsets of all the operations. Given a set O of operations, a process p and an object x, the notation $O|p$ denotes the subset of O that are operations invoked by p, $O|x$ denotes the operations that are applied to x, and $O|w$ denotes those that are write operations. Subsection 2.2 defines the three memory consistency models used in this paper.

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

2.2 Three Memory Consistency Models

This section presents the three widely cited memory consistency models, sequential consistency, P-RAM, and PC-G, and an architecture that gives rise to each[4].

The weakest of these, P-RAM, due to Lipton and Sandberg [12], captures the basic architecture of a fully-replicated, fully-distributed memory. Architectures that implement the P-RAM model are very common in distributed systems [17]. An example architecture with four processes is given in Figure 1. Processes are connected by direct, reliable, FIFO channels, such as TCP connections. Each process has a complete copy of memory. A process reads from its local copy of memory. A write is applied to the local copy, and is propagated to all other processes, which apply it lo-

[4]The framework of Subsection 2.3 can be used to prove that these intuitions are correct (see [9, 5] for the techniques).

cally. The propagation of writes to the processes could cause writes to arrive in different orders at different processes [14].

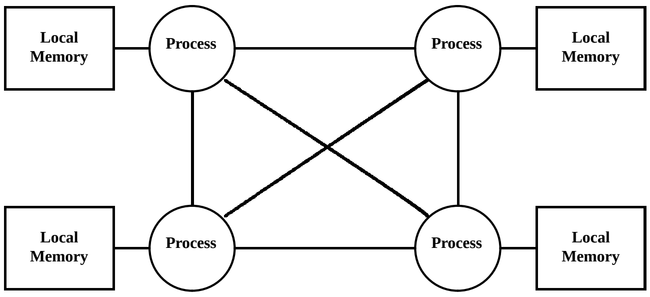

Figure 1: An implementation of P-RAM with four processes.

P-RAM consistency orders operations from each process' point of view. Each process only "sees" all of its own operations and other processes' writes. In P-RAM there must be, for each process, a linear ordering of its operations and all others' write operations that maintains program order and is valid. This does not mean that all processes will order all operations in the same way since the interleaving of the operations by different processes can be perceived differently by each process. The following definition is based on that of Ahamad *et. al.*[1] but reformulated using our framework.[5]

Definition 2.1 *Let O be all the operations of a computation C of a multiprocess program (P,J). Then C is* P-RAM *if for each process $p \in P$ there is a valid total order $(O|p \cup O|w, \xrightarrow{L_p})$ satisfying $(O|p \cup O|w, \xrightarrow{prog}) \subseteq (O|p \cup O|w, \xrightarrow{L_p})$.*

Sequential consistency requires a single linear ordering of all the operations that agrees with each process's view.

Definition 2.2 *Let O be all the operations of a computation C of a multiprocess program (P,J). Then C is* sequentially consistent, *abbreviated* SC, *if there is a valid total order $(O, \xrightarrow{L})$ such that $(O, \xrightarrow{prog}) \subseteq (O, \xrightarrow{L})$.*

An equivalent definition of sequential consistency is [9]:

Definition 2.3 *Let O be all the operations of a computation C of a multiprocess program (P,J). Then C is* sequentially consistent *if for each process $p \in P$ there is a valid total order $(O|p \cup O|w, \xrightarrow{L_p})$ satisfying:*

$$\text{1. } (O|p \cup O|w, \xrightarrow{prog}) \subseteq (O|p \cup O|w, \xrightarrow{L_p}) \text{ and 2. } \forall q \in P, (O|w, \xrightarrow{L_p}) = (O|w, \xrightarrow{L_q}).$$

From the second definition it is clear that sequential consistency requires P-RAM consistency plus agreement on the order of writes by all processes. Such agreement can be achieved in a P-RAM machine by requiring the processes of Figure 1 to perform their writes in a critical section. One

[5]Other interpretations of the original intentions of Lipton and Sandberg are also possible [19].

(inefficient) way to achieve this is to circulate a token on a logical ring of all the processes.[6] To perform a write, a process must wait for the token. The process passes the token after receiving acknowledgements from all other processes that its write has been applied everywhere.

In addition to P-RAM, processor consistency [2] requires that writes to the same variable are seen by all processes in program order. The following definition is due to Ahamad *et. al.*[1].

Definition 2.4 *Let O be all the operations of a computation C of a multiprocess program (P,J). Then C is* PC-G *if for each process $p \in P$ there is a valid total order $(O|p \cup O|w, \xrightarrow{L_p})$ satisfying:*

1. $(O|p \cup O|w, \xrightarrow{prog}) \subseteq (O|p \cup O|w, \xrightarrow{L_p})$ *and 2.* $\forall q \in P$ *and* $\forall x \in J, (O|w \cap O|x, \xrightarrow{L_p}) = (O|w \cap O|x, \xrightarrow{L_q})$.

PC-G can be obtained from the architecture of Figure 1 by creating, for each variable, a logical ring with a circulating token. Writes are performed similarly to the sequentially consistent implementation except there is a token for each variable. A write by process p to variable x can be performed only when p receives the token for x, thus guaranteeing Condition 2 of Definition 2.4. Notice that for single-writer variables, Condition 2 follows from the preservation of program order guaranteed by Condition 1, so circulating tokens are only required for multi-writer variables.

The simulation of sequential consistency on a P-RAM machine requires one critical section in which processes can perform their writes. However, the PC-G simulation requires as many critical sections as variables, allowing two processes to engage in their critical sections simultaneously as long as they are writing different variables. Hence, for a given program, P-RAM admits more computations than both PC-G and sequential consistency, and PC-G admits more computations than sequential consistency.

2.3 Transformations, Interpretations, Implementations and Compilers

Call a collection of objects and a memory consistency model a *platform*. To investigate the possibility of implementing one system on a weaker platform, we first make precise what it means to transform one system, called the *specified system* (with corresponding *specified* objects, processes and memory consistency) into another system that uses a different platform, called the *target platform* (with corresponding *target* objects, and memory consistency). See Figure 2. An operation on a specified object is *transformed* to the target platform by providing a subroutine for the operation invocation that uses only operation invocations on the target objects. If the specified operation invocation has an accompanying response, then the subroutine must return a value of the same type as this response. An *object is transformed* to the target platform by transforming each of its operations. A transformation of all the objects of the specified system to the target objects can be naturally extended to a *process transformation* by replacing each of its operation invocations with the subroutine for that operation. The transformed processes together with the target platform comprise the *transformed system*. The transformed system gives rise to a collection of computations — exactly those that arise from the transformed processes interacting with the target objects and satisfying the target memory consistency constraints. Any such computation can be *interpreted* as a computation of the specified program by attaching to each operation invocation of the specified program, the value returned by the corresponding subroutine.

[6]Another way is to use timestamps to implement what Tanenbaum and Steen call totally-ordered multicasting [17].

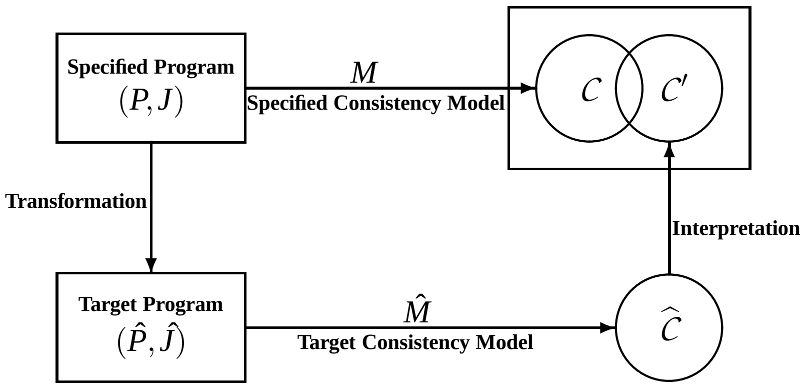

Figure 2: Program transformation and computation interpretation (If $\mathcal{C}' \subseteq \mathcal{C}$, then the transformation is an implementation.)

A transformation of a specified program will be called an implementation of the specified system, if, informally, in Figure 2, the set $\mathcal{C}'$ of computations produced by traveling the long way around is a non-empty subset of the computations $\mathcal{C}$ allowed by the specification. More precisely, let $\mathcal{C}$ be the set of computations of a multiprocess system (P,J,M). Let $\hat{P}$ be the processes of a transformation of (P,J,M) to a platform $(\hat{J},\hat{M})$. Let $\hat{D}$ be any computation of system $(\hat{P},\hat{J},\hat{M})$ and let D' be the interpretation of $\hat{D}$ for (P,J). Then the transformation is an *implementation of the system* (P,J,M) *on the platform* $(\hat{J},\hat{M})$ if $D' \in \mathcal{C}$ for any such D'. If, for every set of processes P consistent with J a given transformation is an implementation of the system (P,J,M) on the platform $(\hat{J},\hat{M})$, then that transformation is a *compiler from platform* (J,M) *to platform* $(\hat{J},\hat{M})$.

For the rest of this paper, operations on specified objects are enclosed in angle brackets; operations on target objects are not bracketed.

3 Compiling SC Platforms to PC-G Platforms

Our goal is to provide a compiler that transforms sequentially consistent systems with read/write shared variables to equivalent systems with only read/write variables and PC-G consistency (henceforth called a *PC-G platform*). Subsection 3.1 achieves this for any 2-process sequentially consistent system provided its shared objects are all single-writer variables. The compiler is extremely simple and uses only one additional variable — a bounded multi-writer variable that is written (but never read!) by each of the two processes. In fact, the multi-writer variables need have only one possible value, though for clarity we use two values. We have been unable to find such a general compiler for more than two processes. However, as shown in Subsection 3.2, if there is one it must be substantially more involved than the simple one that works for two processes.

3.1 Two-Process Case

Transformation: Let J be any collection of shared single-writer variables. We construct a transformation of any 2-process program $P = \{p,q\}$ that is consistent with J to a PC-G platform with one additional variable, m, which is a multi-writer variable. Let y denote either process in

$\{p,q\}$. The (oblivious) transformation α for any single-reader read/write variable s_y that is written by process y is defined by:

Transformation 1 $$\begin{cases} \alpha(\langle \text{write}(s_y,\nu)\rangle) & = & \text{write}(m,id(y)),\text{write}(s_y,\nu),\text{write}(m,id(y)) \\ \alpha(\langle \text{read}(s_y)\rangle) & = & \text{read}(s_y) \end{cases}$$

That is, each write invocation by y is replaced by a sequence of three writes, a write of the process id to m, the original write, and another write of the process id to m. Read invocations are not changed. The extension of this transformation to a process y is denoted $\alpha(y)$. The transformed program $(\hat{P},\hat{J})$ is constructed by setting $\hat{J} = J \cup \{m\}$ and $\hat{P} = \{\alpha(p),\alpha(q)\}$. Any computation of $(\hat{P},\hat{J})$ can be interpreted as a computation of (P,J) by simply ignoring all the writes to m.

Theorem 3.1 shows that any computation of $(\hat{P},\hat{J},PC\text{-}G)$ has an interpretation as a computation of (P,J,SC). That is α is a compiler from sequentially consistent platforms with single-writer variables to PC-G platforms, under the restriction that it is applied to 2-process programs.

Theorem 3.1 *Let J be any collection of single-writer variables. Let $P = \{p,q\}$ be any 2-process program consistent with J. Then the transformation α defined by Transformation 1, is a compiler from the sequentially consistent system (P,J,SC) to the PC-G platform $(J \cup \{m\},PC\text{-}G)$ where m is a two-valued multi-writer variable shared by $\alpha(p)$ and $\alpha(q)$.*

Proof: Let y denote either of $\alpha(p)$ or $\alpha(q)$ (chosen arbitrarily) and $\bar{y}$ denote the other. Denote the single writer variables written by process y by s^i_y for i from 1 to the number of these variables. Any write operation σ to s^i_y is enclosed in the program of y by a *matching* pair of writes to m. For any such pair, denote the preceding write by $\mu^{pre}_y(\sigma)$ and the following write by $\mu^{post}_y(\sigma)$.

Let C be a PC-G computation of the system $(\{\alpha(p),\alpha(q)\},J \cup \{m\})$, and let L_y and $L_{\bar{y}}$ be the valid total orders that are guaranteed by PC-G for y and $\bar{y}$ respectively. Our goal is to construct a single sequence $\mathcal{L}$ containing all the operations of C that is valid and maintains program order.

According to the definition of PC-G, the order of all writes to m are the same in both L_y and $L_{\bar{y}}$ and this order preserves program order. Denote this order by μ_1, μ_2,. . ., and call the subsequences of L_y or $L_{\bar{y}}$ from μ_i to μ_{i+1} the *ith segments*. (The 0th segments are the sequences of operations before the first writes to m.) Form a sequence, S, on all the read operations and all the operations to m in C by simply concatenating[7] the read subsequence in the kth segment of L_y with the read subsequence in the kth segment of $L_{\bar{y}}$ and then concatenating these combined segments in order with the corresponding writes to m separating successive combined segments. That is, let $R_y(k)$ denote the subsequence of all the reads in the kth segment of L_y. Then $S = R_y(0),R_{\bar{y}}(0),\mu_1,R_y(1),R_{\bar{y}}(1),\mu_2,\ldots$

For any write, σ, to the single-writer variable s^i_y, the subsequence of S from $\mu^{pre}_y(\sigma)$ to $\mu^{post}_y(\sigma)$ is denoted $I(\sigma)$. Construct $\mathcal{L}$ by inserting into S all writes to any s^i_y or $s^i_{\bar{y}}$. Each single-writer operation, say σ_y, is placed within the interval $I(\sigma_y)$ of S as follows. If there is any read of the

[7]If a process is allowed to read its own single-writer variable, a more careful merge must be done to ensure that reads of old values precede reads of new values. This is straightforward. However, we can always assume that a process does not read the variable that it writes, but instead keeps a local copy. This simplify the proof slightly.

value written by σ_y in $I(\sigma_y)$ then σ_y is placed immediately before the first such read. Otherwise, σ_y is placed immediately before $\mu_y^{post}(\sigma_y)$. Observe the following about $\mathcal{L}$.

all operations S contains all operations on m (these are only writes) and all reads. Every write to any s_y^i or $s_{\bar{y}}^i$ was inserted into S to construct $\mathcal{L}$. Thus $\mathcal{L}$ contains all operations of C.

program order L_y and $L_{\bar{y}}$ are in program order and S maintains those orders for its elements. So S maintains program order for all writes to m and for all reads. Again, because L_y maintains program order, for any write σ to any s_y^i, the only operation by y between $\mu_y^{pre}(\sigma)$ and $\mu_y^{post}(\sigma)$ in L_y and hence in S is σ. Therefore, any placement of σ within $I(\sigma)$ is consistent with program order. Thus $\mathcal{L}$ is a total order that extends program order.

validity Consider a read operation r of any $s_{\bar{y}}^i$ by process y that returns the value written to $s_{\bar{y}}^i$ by the write operation σ by $\bar{y}$. Also, let $\hat{\sigma}$ be any other write to $s_{\bar{y}}^i$ that follows σ in the program order of $\bar{y}$. Then r must occur after σ and before $\hat{\sigma}$ in L_y because L_y is valid. Since L_y preserves program order, r must therefore occur after $\mu_{\bar{y}}^{pre}(\sigma)$ and before $\mu_{\bar{y}}^{post}(\hat{\sigma})$ in L_y. The order of L_y for all reads and all writes to m is maintained by S and hence by $\mathcal{L}$. Therefore, r must occur after $\mu_{\bar{y}}^{pre}(\sigma)$ and before $\mu_{\bar{y}}^{post}(\hat{\sigma})$ in $\mathcal{L}$. Because of where σ was inserted into S, r must occur after σ in $\mathcal{L}$. Furthermore, r could also occur after $\hat{\sigma}$ in $\mathcal{L}$ only if there was another read $\hat{r}$ of $s_{\bar{y}}^i$ by y that preceded r and returned the value of $\hat{\sigma}$, causing $\hat{\sigma}$ to be inserted before $\hat{r}$. But then in L_y, $\hat{r}$ must also precede r . So in L_y, $\hat{r}$ and r return values of $s_{\bar{y}}^i$ in the opposite order to the order in which these values were written to $s_{\bar{y}}^i$ according to program order of $\bar{y}$. This is impossible because L_y is valid and maintains program order. Thus r must occur in $\mathcal{L}$ after σ and before $\hat{\sigma}$. Therefore, $\mathcal{L}$ is valid.

Since $\mathcal{L}$ contains all the operations of C, extends program order and is valid, it is a valid total order of the operations of C. Hence, by Definition 2.2, C is sequentially consistent, and so is its interpretation for the system (P, J, SC). ■

A few observations are in order. Theorem 3.1 establishes that the interpretation of every PC-G computation of the transformed system is a possible sequentially consistent computation of the original system. We would like to conclude that the transformed system meets the same specification as the original system. Progress is clearly preserved because a computation cannot be stalled or changed by the addition of writes that are never read. Some care is needed to guarantee that safety is preserved. Specifically, for deterministic programs, we need to check that for any computation of the transformed program, its sequentially consistent interpretation was actually permitted by the original system. This holds as long as the original program was assumed to run under an arbitrary scheduler. If, however, the scheduler was somehow constrained by additional assumptions of the specified platform, it is conceivable that the transformed program will produce a computation that, though sequentially consistent, could not have happened under the constrained scheduler. For randomized programs, we also need to check that the probabilities of computations have not been altered. Suppose, for example, in the specified system it is assumed that the scheduler cannot intervene between a random choice and the next operation. Then, in the transformed system, the addition of extra writes would allow the scheduler to intervene, and thus gain some power that it did not originally have. However, as long as the assumptions of the original system assumed both an arbitrary scheduler and atomic random choices (as is generally assumed for sequential consistency) then Theorem 3.1 provides the desired correctness. These conditions are met in our application of the theorem in Section 4. Also, because α is simple straight line code, it preserves properties of the original program. For example, the transformed program is wait-free if the original is.

The code produced by the compiler α could be inefficient because the accesses to the multi-writer variable must appear to be sequential. However, it describes a general method that can be applied to any two-process program. Further optimization of the number of introduced writes could be achieved by analyzing the properties of the specific program to which it is applied.

3.2 Three or more processes

Transformation 1 is not a compiler for more than 2 processes. Theorem 3.2 shows that this failure is quite general. Call a transformation *write-adding* if all it does is insert additional write operations (to new or existing variables) into the original program.

Theorem 3.2 *No write-adding program transformation is a compiler from sequentially consistent systems to PC-G platforms with three or more processes.*

Proof: Suppose there is such a compiler, and suppose it introduces writes to more than one new variable. Since these writes are never read, the only way they can constrain the set of possible PC-G computations is because all processes must agree on the order of operations to each of these variables. By replacing all these writes with writes to one new muliwriter, the set of possible computations is further constrained. Thus, there is also a write-adding compiler, say α, that uses only one new multi-writer variable. Observe that any write-adding transformation that adds writes that are a superset of the writes added by α is also an implementation, because, again this can only further constrain the set of computations and does not eliminate any computation that happens to be sequentially consistent. We conclude that if there is a write-adding compiler, then the transformation α^* that inserts a write to a single multi-writer variable, say m, before and after every operation must be one. It remains only to show that α^* fails to implement some program.

Program 1 is a simple 3-process program $(\{p,q,r\},\{s_p,s_q\})$.

$$\textbf{Program 1} \begin{cases} p: & \text{write}(s_p, \mathit{1}); \text{read}(s_q) \\ q: & \text{write}(s_q, \mathit{2}) \\ r: & \text{read}(s_q); \text{read}(s_p) \end{cases}$$

Computation 1 is a computation of the transformation by α^* of Program 1 assuming s_p and s_q are initially 0. For clarity the writes to m are called pre-write and post-write with each matching pair writing a distinct value.

$$\textbf{Computation 1} \begin{cases} \hat{p}: & \text{pre-write}_p(m, \mathit{p1}); \text{write}(s_p, \mathit{1}); \text{post-write}_p(m, \mathit{p1}); \\ & \text{pre-write}_p(m, \mathit{p2}); \text{read}(s_q) = 0; \text{post-write}_p(m, \mathit{p2}) \\ \hat{q}: & \text{pre-write}_q(m, \mathit{q1}); \text{write}(s_q, \mathit{2}); \text{post-write}_q(m, \mathit{q1}) \\ \hat{r}: & \text{pre-write}_r(m, \mathit{r1}); \text{read}(s_q) = 2; \text{post-write}_r(m, \mathit{r1}); \\ & \text{pre-write}_r(m, \mathit{r2}); \text{read}(s_p) = 0; \text{post-write}_r(m, \mathit{r2}) \end{cases}$$

Consider the following orders for operations of Computation 1

$$\begin{aligned}
(O|p \cup O|w, \xrightarrow{L_p}) &= (\text{pre-write}_p(m,p1); \text{pre-write}_q(m,q1); \text{write}(s_p,1); \text{pre-write}_r(m,r1);\\
&\quad \text{post-write}_r(m,r1); \text{pre-write}_r(m,r2); \text{post-write}_r(m,r2); \text{post-write}_p(m,p1);\\
&\quad \text{pre-write}_p(m,p2); \text{read}(s_q) = 0; \text{post-write}_p(m,p2); \text{write}(s_q,2);\\
&\quad \text{post-write}_q(m,q1))\\
(O|q \cup O|w, \xrightarrow{L_q}) &= (\text{pre-write}_p(m,p1); \text{pre-write}_q(m,q1); \text{write}(s_p,1); \text{pre-write}_r(m,r1);\\
&\quad \text{post-write}_r(m,r1); \text{pre-write}_r(m,r2); \text{post-write}_r(m,r2); \text{post-write}_p(m,p1);\\
&\quad \text{pre-write}_p(m,p2); \text{post-write}_p(m,p2); \text{write}(s_q,2); \text{post-write}_q(m,q1))\\
(O|r \cup O|w, \xrightarrow{L_r}) &= (\text{pre-write}_p(m,p1); \text{pre-write}_q(m,q1); \text{write}(s_q,2); \text{pre-write}_r(m,r1); \text{read}(s_q) = 2;\\
&\quad \text{post-write}_r(m,r1); \text{pre-write}_r(m,r2); \text{read}(s_p) = 0; \text{post-write}_r(m,r2)\text{write}(s_p,1);\\
&\quad \text{post-write}_p(m,p1); \text{pre-write}_p(m,p2); \text{post-write}_p(m,p2); \text{post-write}_q(m,q1))
\end{aligned}$$

Each of these orders contains all write operations and the reads of one process. Each order is valid. Each has the same ordering on writes to m. Each maintains program order. So Computation 1 satisfies PC-G.

The interpretation of Computation 1 as a computation of $(\{p,q,r\},\{s_p,s_q\})$ is Computation 2, which is clearly not sequentially consistent.

$$\textbf{Computation 2} \begin{cases} p: & \text{write}(s_p,1);\ \text{read}(s_q) = 0 \\ q: & \text{write}(s_q,2) \\ r: & \text{read}(s_q) = 2;\ \text{read}(s_p) = 0 \end{cases}$$

■

We conclude that there is no write-adding compiler to PC-G platforms for even very restricted 3-process systems. So any potential compiler for more than 2 processes, even for just single-writer variables, must read some multi-writer variables. Furthermore, a similar proof technique can be used to also eliminate the possibility of an implementation that both reads and writes a single additional multi-writer variable.

Theorem 3.3 *No program transformation that uses only one additional variable and any number of reads and writes is a compiler from sequentially consistent systems to PC-G platforms.*

4 Expected Wait-Free Test-and-Set

Since a Test&Set object has consensus number two [3], Test&Set has no deterministic waitfree implementation using read/write variables even with sequentially consistent memory. In this section, the transformation of Section 3 is used to implement a randomised wait-free Test&Set program in a system that is not sequentially consistent, namely PC-G, using only read/write variables. This is the first (expected) wait-free Test&Set implementation for a weak consistency model. First we specify a Test&Set multiprocess system.

A *Test&Set object*, y, has initial value 0 and supports the operations: $\langle$test&set(y)$\rangle = b$, which returns the value $b \in \{0,1\}$, and $\langle$reset(y)$\rangle$ which set the value of y to 0. A sequence of $\langle$test&set$\rangle$ and $\langle$reset$\rangle$ operations to y, is valid if the first operation in the sequence and the first operation after each $\langle$reset(y)$\rangle$ is $\langle$test&set(y)$\rangle= 0$ and all other $\langle$test&set(y)$\rangle$ operations are $\langle$test&set(y)$\rangle= 1$. For the operation $\langle$test&set(y)$\rangle= b$ the invocation component is $\langle$test&set(y)$\rangle$ and the response is b. The response to the operation invocation $\langle$reset(y)$\rangle\}$ is an acknowledgement.

A *Test&Set* process, p has the code structure:

```
repeat
    if ⟨test&set(y)⟩ = 0 then
        ⟨exclusion section(p)⟩
        ⟨reset(y)⟩
    else
        ⟨remainder(p)⟩
    end-if
until done
```

where y is a Test&Set object and the operations of $\langle$remainder(p)$\rangle$ and $\langle$exclusion section(p)$\rangle$ are any operations applied to any objects other than y.

A Test&Set system is a system $(P, \{y\}, SC)$ where P is a collection of Test&Set processes and y is a Test&Set object. The requirement that any computation of the Test&Set system be sequentially consistent, together with the definition of the Test&Set processes and object, is enough to ensure that any interleaving of the operations of the processes into a valid total order, will never have more than one process whose most recent operations was $\langle$test&set(y)$\rangle = 0$. So our system provides what we expect from a Test&Set object (no two processes can "simultaneously" be in $\langle$synchronized section$\rangle$). The definition however is time-independent; it does not rely on a global clock. In addition to the safety requirement, which is captured by the definition of a Test&Set object combined with sequential consistency, the system must also satisfy the *Progress* and *Expected Wait-Freedom* properties.

Progress: Whenever some process, p, has its program counter at $\langle$test&set(y)$\rangle$ eventually, some process q will have its program counter at $\langle$reset(y)$\rangle$ under the assumption that $\langle$exclusion section(p)$\rangle$ is finite. *Expected Wait-Freedom:* With probability one, each process p completes $\langle$test&set(y)$\rangle$ in a finite number of steps.

In this section we show how to implement the Test&Set system on a PC-G platform.

Two-process case

Wait-free Test&Set cannot be deterministically implemented with just read/write variables even on a Linearizable platform [13]. However, Tromp and Vitanyi [18] gave a 2-process randomized algorithm that uses only two single-writer variables. Hence, by Theorem 3.1, Transformation 1 applied to their solution provides an implementation of expected wait-free Test&Set on a PC-G platform.

Figure 3 reproduces Tromp and Vitany's Test&Set algorithm [18] (all non-boxed operations). In this figure, $s_i, s_j \in \{me, you, choose, rst\}$ and random(me,you) denotes a fair coin flip to choose between me and you. The original algorithm corresponds to the program $(\{i, j\}, \{s_i, s_j\})$. The figure also shows the transformation to $(\{\alpha(i), \alpha(j)\}, \{s_i, s_j, m\})$, using Transformation 1.

```
define ⟨test&set($\Omega, i$)⟩
    if ($s_i = you$ and $s_j \neq rst$) then return 1
    repeat
        $\boxed{m \leftarrow i}$
        $s_i \leftarrow choose$
        $\boxed{m \leftarrow i}$
        case $s_j$ is
            $you, rst$: $\boxed{m \leftarrow i}$
                        $s_i \leftarrow me$
            $me$      : $\boxed{m \leftarrow i}$
                        $s_i \leftarrow you$
            $choose$  : $\boxed{m \leftarrow i}$
                        $s_i \leftarrow$ random($me$,$you$)
        end-case
        $\boxed{m \leftarrow i}$
    until ($s_i \neq s_j$)
    if ($s_i = me$) then return 0
    else return 1
```

```
define ⟨reset($\Omega, i$)⟩
    $\boxed{m \leftarrow i}$
    $s_i \leftarrow rst$
    $\boxed{m \leftarrow i}$
```

Figure 3: Waitfree Test-and-Set program for a PC-G platform

The program of process i is shown; process j's is symmetric ($j = 1 - i$). The invocations that are enclosed in boxes correspond to the transformation. Note that each write to the single-writer variable s_i is preceded and followed by a box.

Corollary 4.1 *The program given in Figure 3 is an implementation of expected wait-free Test&Set on a PC-G platform with 2 processes.*

The general case

Since we only require sequential consistency (versus Linearizability) for Test&Set programs, a general solution for any number of processes can be achieved using a classical tournament tree construction. Informally, the processes are arranged at the leaves of a balanced binary tree. A successful 2-process ⟨test&set⟩ (returning 0) advances the winner to the parent node. A successful n-process ⟨test&set⟩ completes at the root. The winning process executes the n-process ⟨reset⟩ by executing a 2-process ⟨reset⟩ for each node along its reversed path from the root to its original leaf. To complete their failed Test&Set operation. losing processes must also execute reset on the reversed path from the last node they won to their original leaf position,

The single-writer variables needed for the 2-process Test&Set are no longer single-writers in the general-case construction. Each process can potentially write variables associated with any node on the path from the process's leaf node to the root. However, by the correctness of the 2-process program, only one "winner" is allowed to advance to a parent node. That is, by the nature of 2-process Test&Set, there cannot be more than one process competing at the same node "simultaneously".

This particular algorithm is an example of a n-process program that uses multi-writer variables that is implemented on a PC-G platform by transformation α even though the same transformation fails for some 3-process single-writer algorithms.

5 Conclusion

The question remains whether there is a “simple” compiler that converts any sequentially consistent program (or a large class of them) to an equivalent one for a PC-G platform. It is possible that reading and writing several multi-writers breaks the impasse, though we conjecture that any such solution will not provide wait-freedom. The partial progress reported in this paper has substantial implications. There are important 2-process programs that use only single-writer variables, which can be compiled for the PC-G platform. Tromp and Vitanyi’s expected wait-free Test&Set is an example.

References

[1] M. Ahamad, R. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In *Proc. 5th Int’l Symp. on Parallel Algorithms and Architectures*, pages 251–260, June 1993. Technical Report GIT-CC-92/34, College of Computing, Georgia Institute of Technology.

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

[3] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.

[4] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.

[5] L. Higham, L. Jackson, and J. Kawash. Specifying memory consistency of write buffer multiprocessors. In Preparation.

[6] L. Higham and J. Kawash. Java: Memory consistency and process coordination (extended abstract). In *Proc. 12th Int’l Symp. on Distributed Computing, Lecture Notes in Computer Science volume 1499*, pages 201–215, September 1998.

[7] L. Higham and J. Kawash. Bounds for mutual exclusion with only processor consistency. In *Proc. of the 14th Int’l Conf. on Distributed Computing, Lecture Notes in Computer Science volume 1914*, pages 44–58, October 2000.

[8] L. Higham and J. Kawash. Memory consistency and process coordination for SPARC multiprocessors. In *Proc. of the 7th Int’l Conf. on High Performance Computing, Lecture Notes in Computer Science volume 1970*, pages 355–366, December 2000.

[9] J. Kawash. *Limitations and Capabilities of Weak Memory Consistency Systems.* Ph.D. dissertation, Department of Computer Science, The University of Calgary, January 2000.

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

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

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

[13] M. Loui and H. H. Abu-Amara. *Advances in Computing Research*, volume 4, chapter Memory Requirements for Agreement Among Unreliable Asynchronous Processes, pages 163–183. JAI Press, 1987.

[14] D. Mosberger. Memory consistency models. *ACM Operating Systems Review*, 27(1):18–26, January 1993.

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

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

[17] A. S. Tanenbaum and M. V. Steen. *Distributed Systems Principles and Paradigms*. Prentice Hall, 2002.

[18] J. Tromp and P. Vitányi. Randomized two-process wait-free test-and-set. *Distributed Computing*, 15:127–135, 2002.

[19] N. Verwaal. Ambiguous memory consistency models. Master's thesis, Department of Computer Science, The University of Calgary, 1998.