Inferring Fences in a Concurrent Program Using SC proof of Correctness

Chinmay Narayan, Shibashis Guha, S.Arun-Kumar
Indian Institute of Technology Delhi

Abstract. Most proof systems for concurrent programs [Jon83] [O’H07] [VP07] assume the underlying memory model to be sequentially consistent (SC), an assumption which does not hold for modern multicore processors. These processors, for performance reasons, implement relaxed memory models. As a result of this relaxation a program, proved correct on the SC memory model, might execute incorrectly. To ensure its correctness under relaxation, fence instructions are inserted in the code. In this paper we show that the SC proof of correctness of an algorithm, carried out in the proof system of [Sou84], identifies per-thread instruction orderings sufficient for this SC proof. Further, to correctly execute this algorithm on an underlying relaxed memory model it is sufficient to respect only these orderings by inserting fence instructions.

1 Introduction

The memory model of a processor defines the order in which memory operations, issued by a single processor, appear to execute from the point of view of the memory subsystem. In a broad sense, it determines whether any two memory access instructions issued by a processor within a single thread can be reordered. Sequentially Consistent memory model (SC) is the simplest but most restrictive of all and does not allow any reordering of instructions within a thread. Modern multicore processors, for the purpose of hiding latencies, implement relaxed memory models and allow instructions within a thread to be reordered as long as they operate on different memory addresses. For example, Total Store Order (TSO), the memory model for x86 processors, allows write instructions to get reordered with later reads provided they operate on different memory locations. As a result of these relaxations, a program may exhibit more behaviours than under SC and it is possible that some of these extra behaviours do not satisfy the property which holds under SC. Peterson’s mutual exclusion algorithm, in Figure 1 illustrates this behaviour. This algorithm satisfies mutual exclusion property under the SC model but executing it on an Intel’s x86 processor might result in the violation of this property. This can happen if the read of flag2 at label 3 is reordered before instructions at label 1 and label 2. With such reordering Proc1 can enter the critical section. Proc2, with or without this reordering, can also enter the critical section simultaneously. This example clearly shows the effect of memory model on the correctness of an algorithm.
It is clear that this problem appeared because of an extra execution generated due to instructions’ reordering which was not possible under the SC memory model. This problem also does not appear for a data race free program if the underlying relaxed memory model satisfies data-race freedom (DRF) property.

It can be shown that any data race free program when executed on a memory model satisfying DRF exhibits exactly the same set of behaviours as under the SC memory model. A program is data race free if in every execution of this program any pair of an conflicting instructions (w-w or w-r) by two different threads to the same variable are separated by an unlock instruction. Therefore a data race free and a correct program under SC is guaranteed to execute correctly on a memory model which satisfies DRF property. However, the algorithms that we are interested in (lock-free, wait-free, lock implementations) do not use lock/unlock and hence do not fit under this definition of data race freedom. Therefore, their correctness under a relaxed memory model does not follow from their correctness under the SC memory model.

Another way to avoid extra executions is to prevent certain reorderings by putting a special instruction, fence, after every instruction in each thread. A fence instruction when placed between any two instructions in a thread prohibits their reordering. Execution of a fully fenced program (fence after every instruction in a thread) on a relaxed memory model generates exactly the same set of executions as under SC and therefore the correctness under SC implies the correctness under relaxed memory model. However, this trivial placement strategy would negate the performance benefits associated with relaxed memory models. Therefore, an ideal placement of fence instructions should preserve only those program orders which are sufficient to prove the correctness of the properties of interest.

In this paper, we deal with parallel programs which satisfy some property under SC but are not race-free. The main contribution of this paper is to show that the proof of correctness of these programs under SC is useful in identifying per-thread instruction orderings sufficient to make this program correct on a relaxed memory model. Further, locations of fence instructions can be inferred based on these orderings and the underlying memory model. We are not aware of any other attempt to use the SC proof of correctness for fence inference in a concurrent program.
2 Related Work

All existing approaches for inferring fences for relaxed memory models can be divided into two main categories; model checking based approaches [HR07] [KVY11] [LW11] [AAC+12b] and proof system based approaches [Rid10] [Bor12] [BD12]. Model checking based approaches first explore the state space of a program under a given memory model using buffer based operational semantics and check the reachability of erroneous states. Once a reachable erroneous state is identified, the path leading to this state is restricted by inserting fences at appropriate places. [ABBM12] and [ABBM10] showed that the state reachability problem for TSO and PSO memory models is decidable for finite state programs. Further, this problem becomes undecidable as soon as the read after write reordering is added to the memory model. This approach, by its nature, is better suited to programs with finite data domains. We are aware of only one line of work [AAC+12a] which combines predicate abstraction and model checking based approach to verify and correct infinite data domain programs like Lamport’s bakery algorithm.

The second approach is to use a memory model specific proof system as done in [Rid10]. [Rid10] presents a separation logic based proof system for the TSO memory model and shows that Simpson’s 4 slot algorithm does not satisfy the interference freedom property. Recently [Bor12] and [BD12] looked at the use of separation logic derived proof system for verifying concurrent data structures on POWER/ARM based memory models. These memory models are more complex than the TSO or the PSO memory model mainly because of non-atomic writes. Unlike the approaches of [Rid10], [Bor12] and [BD12], we do not propose a memory model specific proof system but only look at the proof of correctness under SC memory model and use it to infer sufficient orderings required for the correctness. Unlike model checking approaches, proof system based approaches can cover more than just reachability. This is evident in the example of Simpson’s 4 slot algorithm where apart from the interference freedom we also prove that the sequence of values observed by the reader are consistent, i.e. they form a stuttering sequence of the values written by the writer. We are not aware of any line of work which handled the fence inference in Simpson’s 4 slot algorithm under PSO memory model with respect to the interference freedom and the consistent reads properties.

3 Language: Syntax and Semantics

Figure 2 shows the syntax of a simple parallel programming language without the support of dynamic thread creation. Operator $\parallel$ is used to compose a finite number of programs in parallel. We explicitly distinguish local variables, ranged over by $\ell\text{var}$ and accessed only within a thread, and shared variables, ranged over by $\text{shvar}$ and accessed by more than one thread. A local expression, ranged over by $\ell\text{exp}$, is constructed using only local variables, values and operators. A shared expression, ranged over by $\text{shexp}$, is constructed from exactly one shared variable, another local expression and operators. Assignment command only allows
assigning a local expression to a local variable, a shared expression to a local variable or a local expression to a shared variable. Assignment of a shared expression to a shared variable can be broken down into assignments of one of the above forms. Because of this restriction every assignment command either reads at most one shared variable or writes to at most one shared variable but not both. This guarantees at most one memory load or store event per assignment expression which is helpful in reasoning about memory model and associated events.

3.1 SC Semantics

Figure 3 shows the semantics of this language under SC. A state or configuration under SC is of the form \((G, Tstore)\) where

\[
G \overset{def}{=} \text{shvar } \rightarrow \text{val}, \quad Tstore \overset{def}{=} \text{Tid } \rightarrow \mathcal{L} \times \text{Program}, \quad \mathcal{L} \overset{def}{=} \text{ℓvar } \rightarrow \text{val}
\]

Local store \(\mathcal{L}\) and global store \(G\) maps local and shared variables respectively to their values. Each thread is represented by a unique thread id. Thread store maps a thread id to its local store and the program to be executed next. In our operational semantics we use the set representation of this function, i.e \(Tstore\) as a set of tuples of the form \((t, \mathcal{L}, C)\). Function \([\cdot]_{G, \mathcal{L}} \in \text{exp } \rightarrow G \rightarrow \mathcal{L} \rightarrow \text{val}\) such that

\[
[G(\text{shvar})]_{G, \mathcal{L}} = G(\text{shvar}), \quad [\text{ℓvar}]_{G, \mathcal{L}} = \mathcal{L}(\text{ℓvar}), \quad [e_1 + e_2]_{G, \mathcal{L}} = [e_1]_{G, \mathcal{L}} + [e_2]_{G, \mathcal{L}}
\]

takes an expression and evaluates it to a value based on the mapping of variables in global store \(G\) and thread local store \(\mathcal{L}\). This function is then used to define reference semantics in Figure 3. For any function \(\mathcal{F} : A \rightarrow B, a' \in A, b' \in B\), we write \(\mathcal{F}[a' := b']\) to denote a function which is same as \(\mathcal{F}\) everywhere except at \(a'\) where it evaluates to \(b'\). Semantic rules corresponding to the conditional and the looping constructs (ITE-T, ITE-F, WHL-T, WHL-F), local variable’s read write (LRW) and global variable’s read write (GR, GW) are quite straightforward. In the parallel composition command, the parent thread stops its execution and
A sequence of elements of the form (\(?\)shvar := \(\ell \)exp; \(\ell \)exp) shared variables in terms of values read from and written to them.

In the proof system of [Sou84] every process (Proc \(_i\)) proceeds as follows: (Fig. 2 one process 

Therefore we sometime use \(\text{join}()\) command in the parent thread for each spawned thread. The rule for \(\text{join}(\text{tid})\) command ensures that this command is executed when the thread corresponding to \(\text{tid}\) has finished its execution. This also ensures that the parent thread waits for the completion of children threads before continuing further.

We add \(\text{fence}\) and \(\text{skip}\) commands like no-op under SC semantics. Following the syntax of Figure 2 one process \(\text{Proc}_i\), or thread, can only execute one program \(\text{Program}_i\). Therefore we sometime use \(\text{Proc}_i\) and \(\text{Program}_i\) interchangeably to mean the same thing.

4 Logic

In the proof system of [Sou84] every process \(\text{Proc}_i\) executing a program \(\text{Program}_i\) has a history variable \(h_{\text{Proc}_i}\) which captures the interaction of this process with shared variables in terms of values read from and written to them. \(h_{\text{Proc}_i}\) is a sequence of elements of the form \((?\text{shvar}, \text{ph})\) or \((!\text{shvar}, v)\). \((?\text{shvar}, \text{ph})\) is
added to the sequence when \( \text{Proc}_i \) reads a value from the shared variable \( \text{shvar} \). Similarly, \((!\text{shvar}, v)\) is added to the sequence when \( \text{Proc}_i \) writes a value \( v \) to the shared variable \( \text{shvar} \). Local reasoning of a program \( \text{Program}_i \) generates a triple of the form \( \{P_i, Q_i, \text{shexp}_i\} \) where \( P_i \) and \( Q_i \) define assertions on the local state of the process as well as on the history \( h_{\text{Proc}_i} \). Rest of this section describes the axioms of this proof system explaining the idea of local reasoning in terms of history variable and the parallel composition rule in terms of \( \text{Compat} \) predicate.

**Axioms** In our programming language all program constructs, except \( \text{GR} \) and \( \text{GW} \), operate on local expression and therefore do not require reading from or writing to shared variables. Therefore the proof rules for these constructs are same as the Hoare’s axioms in sequential setting. In the following proof rules, the notation \( P[Q/Q'][R/R'] \) denote simultaneous substitution of \( Q' \) for \( Q \) and \( R' \) for \( R \) in \( P \). Operator “\( \cdot \)” concatenates an element to a sequence and \( \epsilon \) is the empty sequence.

Given a sequence \( \sigma \), \( |\sigma| \) is the length and \( \sigma[i] \) is the \( i^{th} \) element of this sequence. The proof rule for the assignment to shared variable is as following,

\[
\{P[h_{\text{Proc}_i}/h_{\text{Proc}_i}](!\text{shvar}, \ell\text{exp})]\} \text{shvar}:=\ell\text{exp} \{P\} \quad \text{(GWrite)}
\]

As a result of this write the history is appended with the element \((!\text{shvar}, \ell\text{exp})\). The proof rule for the reading of a shared variable, \( \ell\text{var}:=\text{shexp}_j \), is given as

\[
\{\forall\text{ph}. P[h_{\text{Proc}_i}/h_{\text{Proc}_i}, (?\text{shvar}_j, \text{ph})](\ell\text{var}/\text{shexp}_j[\text{shvar}_j/\text{ph}])\} \ell\text{var}:=\text{shexp}_j \{P\} \quad \text{(GRead)}
\]

This rule requires the value of the shared variable \( \text{shvar}_j \) in order to evaluate the expression \( \text{shexp}_j \). But while reasoning locally we do not know the value beforehand. Therefore instead of the actual value of \( \text{shvar}_j \) a placeholder variable \( \text{ph} \) is assigned to \( \text{shvar}_j \) and this information is stored in \( h_{\text{Proc}_i} \) by appending it with \((?\text{shvar}_j, \text{ph})\). Further, \( \text{shexp}_j \) is evaluated accordingly before being assigned to \( \ell\text{var} \) in the assertion. Here \( \text{ph} \) is universally quantified to all possible values.

\[
\{P_i \land h_{\text{Proc}_i} = \epsilon\} \text{Program}_i \{Q_i\}, \quad i = 1 \cdots n \\
\{P_1 \land \cdots \land P_n \land \text{shvar}_1 = v_1 \land \cdots \land \text{shvar}_m = v_m\} \\
\text{Program}_1 || \cdots || \text{Program}_n \\
\{Q_1 \land \cdots \land Q_n \land \text{Compat}(v_1, \cdots, v_m, h_{\text{Proc}_1}, \cdots, h_{\text{Proc}_n})\} \quad \text{(PARCOMP)}
\]

In parallel composition, each process is analyzed in isolation with the initial value of its history variable \( h_{\text{Proc}_i} \) set to empty and some precondition \( P_i \) on its local state. This gives post-condition \( Q_i \) for each process \( \text{Proc}_i \) which also contains the assertions on \( h_{\text{Proc}_i} \). Precondition of the parallel composition rule is the conjunctions of individual processes’ preconditions and the initial values of shared variables \( \text{shvar}_1 \) to \( \text{shvar}_m \). Post-condition of this rule is the conjunctions of individual processes’ post-conditions along with the predicate \( \text{Compat} \) where \( \text{Compat} \) is defined in Figure 4. \( \text{Merge}(h_{\text{Proc}_1}, \cdots, h_{\text{Proc}_n}) \) represent the set of all possible interleavings of histories \( h_{\text{Proc}_i} \) to \( h_{\text{Proc}_n} \), such that the sequence
Compat$(v_1, \ldots, v_m, h_{proc_1}, \ldots, h_{proc_n}) \overset{\text{def}}{=} \begin{cases} \text{shvar}_1 = v_1 & \text{if } h_{proc_i} = \epsilon, \ i = 1 \cdots n \\ \exists h \in \text{Merge}(h_{proc_1}, \ldots, h_{proc_n}). \\ \forall j \leq m. \text{shvar}_j = f_j(v_j, h) \\ \forall k \leq |h|. h[k] = (\text{shvar}_j, \text{ph}_j) \Rightarrow \text{ph}_j = f_j(v_j, h[1 : k - 1]) \end{cases}$

Fig. 4. Non-recursive definition of Compat predicate

of elements within them is preserved in the merged history. Function $f_j(v, h)$ returns the last value written to the variable shvar$_j$ in history $h$. It returns $v$ if no such write is found. Essentially, the predicate Compat generates a set of equality predicates (one corresponding to each merged history). The first line in Compat’s definition denotes that the final value of any shared variable shvar$_j$ is the last value written to shvar$_j$ in that merged history. Second line relates the placeholder value ph$_j$, corresponding to a read of a shared variable shvar$_j$, to the latest value written to shvar$_j$ just before this element in the merged history. This is sufficient to characterize all compatible merged histories and therefore plays the central role in proofs of §6 and of Appendix ??.

Individual process histories contains placeholder variables for every read. In Figure 5 we define, rather informally, a set of predicates over these histories in order to succinctly represent them in our proofs. Given a sequence $\sigma$, $\sigma|_{\text{type}}$ denote the restricted subsequence of $\sigma$ consisting of only type elements. Predicate None! holds if the history does not contain any write to any shared variable. None!shvar and None?shvar hold if the history does not contain any write to or read from the variable shvar. $[P]^*$ and $[P]^+$ capture the regularity of history sequence by abstracting the placeholder variables. We also admit $P^+ \Rightarrow P^*$ as a relaxation on the history sequence which is used in the consequence rule. We admit that the use of these predicates, without giving proper semantics, is not fully justified but we do it solely for the purpose of making our proofs manageable. We leave more formal treatment of these predicates for the future work.

5 Relaxed Memory Model

For the rest of this paper we consider the PSO memory model which allows reordering of a write instruction with future reads and future write instructions operating on different variables. To simulate the effect of PSO, every thread is equipped with one buffer per shared variable. These buffers store the values written on the corresponding variable by this thread in a queue(FIFO) discipline. Buffering the value of the write in a variable specific queue, simulates the effect of delaying the execution of a write instruction past future reads and future writes on different memory locations. If a read instruction for any shared variable shvar is executed in a thread then the local buffer of shvar is checked first. If this buffer
is non empty then the latest written value (from the tail) is returned. In case of empty buffer the value is read from the global state. This memory model also provides an explicit fence instruction in order to restrict reordering of any two instructions within a thread. Operationally this is achieved by flushing all the buffers of that thread.

For relaxed semantics we need some modifications in the notion of State. A state or configuration under this memory model is of the form \((G, T_{store})\) where,

\[
G \overset{def}{=} \text{shvar} \rightarrow \text{val} \quad T_{store} \overset{def}{=} \text{Tid} \rightarrow \mathcal{L} \times \mathcal{B} \times \text{Program}
\]

\[
\mathcal{B} \overset{def}{=} \text{shvar} \rightarrow \sigma \quad \mathcal{L} \overset{def}{=} \ell_{\text{var}} \rightarrow \text{val}
\]

The only change with respect to SC state is in the definition of thread store. Here, the range of this function also contains a buffer store \(B\) which is a function from shared variable to an ordered sequence of values, ranged over by \(\sigma\). Function \([[-]]_{G,\mathcal{L},\mathcal{B}}^n \in \text{exp} \rightarrow G \rightarrow \mathcal{L} \rightarrow \mathcal{B} \rightarrow \text{val}\), defined in Figure 7, takes an expression and evaluates it based on the values stored in the global store, thread local store and buffer store. Further, relaxed semantics is defined in Figure 6. For the constructs not shown in Figure 6 the semantics is the same as in the SC semantics of Figure 3 except for the change in the evaluation function from \([[-]]_{G,\mathcal{L}}^n \) to \([[-]]_{G,\mathcal{L},\mathcal{B}}^n\). In relaxed semantics, a thread \(t\) executing a write to a shared variable \(\text{shvar}\) enqueues the value in the buffer of \(\text{shvar}\) in \(t\). Any read of a shared variable \(\text{shvar}\) returns the latest value in the buffer of \(\text{shvar}\), if any. If this buffer is empty then the read returns the value from the global state (memory). Flush operation non-deterministically deques an element from any thread buffer and updates the global state accordingly. Further, in the parallel composition rule the requirement for the parent thread’s buffer being empty ensures that instructions before and after the parallel composition are ordered. Same holds for the end command as well.

6 Examples

We use our proof system to prove the correctness of Lamport’s bakery algorithm [Lam74] for two processes and Simpsons’s 4 slot algorithm [Sim90].
assertions about individual elements in a history sequence then

One important notation used in our proofs is as following; Let

Lamport’s algorithm has unbounded data domain for token variables which

- Simpson’s 4 slot algorithm implements a wait-free and lock-free atomic reg-

ister for concurrent reader and writer. This algorithm uses disjoint slots to 
read from and write to in presence of interference. We prove that this algo-

rithm is safe in the sense that concurrent reader and writer never use the 
same slot in presence of interference. Further, we also prove that the values 
observed by successive reads are in the same order as written by the writer.

One important notation used in our proofs is as following: Let \(P\) and \(Q\) be the 
assertions about individual elements in a history sequence then \(P \prec Q\) denotes
the fact that the element satisfying the assertion $P$ appears before the element satisfying the assertion $Q$ in the history sequence.

6.1 Example: Lamport’s Bakery Algorithm for Two Processes

In Lamport’s algorithm each process $Proc_i$ operates on shared variables $token_i$ and $taking_i$ of type integer and boolean respectively. When a process $Proc_i$ intends to enter the critical section, it first reads the value of $token_i$ corresponding to another process, say $v$, and assigns the value $v + 1$ to its own $token_i$ variable. $ftok_1$ and $ftok_2$ are local variables of $Proc_1$ which hold the value of $token_1$ and $token_2$ respectively. Similarly, $stok_1$ and $stok_2$ are local variables of $Proc_2$ for $token_1$ and $token_2$. $(a, b) < (a', b')$ denotes lexicographic less than relation, i.e. $(ftok_2, 2) < (ftok_1, 1)$ iff $ftok_2 < ftok_1$ and $(stok_1, 1) < (stok_2, 2)$ iff $stok_1 \leq stok_2$. This algorithm with inline assertions is shown in Figure 9 where $Inv_1$ and $Inv_2$ are as in Figure 8. Assertions are on the history $h_i$ and local variables of that process. History $h_i$ is abstracted using the predicates of Figure 5. It should be noted that $V_1, W_1$ and $X_1$ do not appear explicitly inside the regular structure of the history abstracted by $[Q_1; R_1; T_1; U_1; \text{None}; W_1; X_1][h]$. They appear in the last iteration of the loop and therefore the variables updated inside the loop, $(ftok_1, ftok_2)$, are assigned the placeholders corresponding to these reads, i.e. $ph_3, ph_4, ph_5$. Same holds for the invariant of $Proc_2$ as well. Subsequently, these elements are abstracted to $\text{None!}$ in order to establish the loop invariant.

**Mutual Exclusion proof** In order to prove the mutual exclusion property, we first state the required assertion to capture this property.

$$ME \overset{df}{=} (Inv_1''(h_1) \land Inv_2''(h_2) \land \text{Compat}(0, 0, false, false, h_1, h_2) = false)$$
where $\text{Inv}1''(h_1)$ and $\text{Inv}2''(h_2)$ are the assertions inside critical sections of $\text{Proc}_1$ and $\text{Proc}_2$ respectively. We prove it in two steps. First we show that,

$$
\text{Inv}1'(h_1) \land \text{Inv}2'(h_2) \land \text{Compat}(0, 0, \text{false}, \text{false}, h_1, h_2) \implies \text{Inter}.
$$

$$
\text{Inter} \overset{\text{def}}{=} \left( \text{Inter}1 \overset{\text{def}}{=} (\text{ftok}_2 = 0 \implies \text{stok}_1 \neq 0 \land \text{ftok}_1 < \text{stok}_1) \right)
\land \left( \text{Inter}2 \overset{\text{def}}{=} (\text{stok}_1 = 0 \implies \text{ftok}_2 \neq 0 \land \text{ftok}_2 < \text{ftok}_1) \right)
\land \left( \text{Inter}3 \overset{\text{def}}{=} (\text{ftok}_2 \neq 0 \land \text{ftok}_1 \neq 0 \implies \text{ftok}_1 \leq \text{ftok}_2 \implies \text{stok}_1 \leq \text{stok}_2) \right)
$$

and then it is easy to show that,

$$
\text{Inter} \land \neg(\text{ftok}_2 \neq 0 \land (\text{ftok}_2, 2) < (\text{ftok}_1, 1))
\land \neg(\text{stok}_1 \neq 0 \land (\text{stok}_1, 1) < (\text{stok}_2, 2)) = \text{false}
$$
Proof of $\text{Inv}1'(h_1) \land \text{Inv}2'(h_2) \land \text{Compat}(0, 0, \text{false}, false, h_1, h_2) \implies \text{Inter}1$, We show that in two steps,

$$\text{Inv}1'(h_1) \land \text{Inv}2'(h_2) \land \text{Compat}(0, 0, \text{false}, false, h_1, h_2) \\
\land ftok_2 = 0 \implies stok_1 \neq 0 \quad (1)$$

and

$$\text{Inv}1'(h_1) \land \text{Inv}2'(h_2) \land \text{Compat}(0, 0, \text{false}, false, h_1, h_2) \\
\land ftok_2 = 0 \implies stok_1 < stok_2 \quad (2)$$

Proof of (1), First assume that $\text{Inv}1'(h_1), \text{Inv}2'(h_2), \text{Compat}(0, 0, \text{false}, false, h_1, h_2)$ and $ftok_2 = 0$ hold. Only way to have $ftok_2 = ph_4 = 0$ is to put last read of $\text{token}_2$ in $h_1$ (denoted by $X_1$) in the merged history after $P_2$ of, say $k-1$th iteration of $[Q_2; R_2; T_2; U_2; \text{None}!; P_2]$ and before $T_2$ of $k$th iteration. $\text{Inv}1$ implies that $T_1 < X_1$ hence the value of $\text{token}_1$ visible at $X_1$ is non-zero which also becomes visible at $T_2$ because of the placement of $X_1$ before $T_2$ and the fact that the history $h_{\text{Proc}_1}$ does not contain any write to $\text{token}_1$ after $X_1$. Further because $\text{Inv}2$ implies $T_2 < W_2$ therefore the read of $\text{token}_1$ at $W_2$ in $\text{Proc}_2$ also observes this non-zero value of $\text{token}_1$ and assigns it to $stok_1$. Therefore we have,

$$\text{Inv}1(h_1) \land \text{Inv}2(h_2) \land \text{Compat}(0, 0, \text{false}, false, h_1, h_2) \\
\land ftok_2 = 0 \implies stok_1 \neq 0$$

Out of all the orderings of $\text{Proc}_1$ and $\text{Proc}_2$ only $T_1 < X_1$ and $T_2 < W_2$ are used to prove this part of the proof. Hence these two are sufficient to prove (1).

Proof of (2) In order to have $ftok_2 = ph_4 = 0$, $X_1$ should be placed in the merged history after $P_2$, of say $k-1$ iteration of $[Q_2; R_2; T_2; U_2; \text{None}!; P_2]$ and before $T_2$ in $k$th iteration. Further, $\text{Inv}1'$ implies $ph_2 = true$ and therefore $V_1$ (corresponding to the read of $\text{taking}_2$ in last iteration) must be placed before $Q_2$ of any $n$th iteration and after $U_2$ of $n-1$th iteration such that $n < k$. It must be noticed that $V_1$ cannot be put after $U_2$ of $k$th iteration because $V_1 < X_1$ in the history of $\text{Proc}_1$. $\text{Inv}1$ implies $T_1 < V_1$ and hence the value of $\text{token}_1$ visible at $V_1$ is non-zero, say $v$. As $V_1$ has been placed before $Q_2$ of $n$th iteration, where $n < k$, hence the same value $v$ of $\text{token}_1$ is also visible at $Q_2$ of $n$th iteration. $\text{Inv}2$ implies that $Q_2 < R_2$ hence the same value $v$ of $\text{token}_1$ is also visible at $R_2$ of any subsequent iterations. $\text{Inv}2$ implies $R_2 < T_2$ and therefore all subsequent iterations of $T_2$ write $v + 1$ to $\text{token}_2$ resulting in $stok_2 = v + 1$. Further the value visible at $stok_1$ in $\text{Proc}_2$ is still the last value written to $\text{token}_1$ by $\text{Proc}_1$, i.e. $v$. Therefore we get,

$$\text{Inv}1'(h_1) \land \text{Inv}2'(h_2) \land \text{Compat}(0, 0, \text{false}, false, h_1, h_2) \\
\land ftok_2 = 0 \implies stok_1 < stok_2$$

Only orderings used in this part of the proof are $V_1 < X_1$, $T_1 < V_1$, $R_2 < T_2$ and $Q_2 < R_2$. Hence, out of all total orders in $h_1$ and $h_2$ these are sufficient to prove (2).
Proof of $Inv 1' \land Inv 2' \land Compat(0, 0, \text{false, false, } h_1, h_2) \implies Inter 2$ , This is symmetric to previous proof and gives us following symmetric sufficient orderings; $T_2 < X_2, T_1 < W_1, V_2 < X_2, T_2 < V_2, Q_1 < R_1$ and $R_1 < T_1$.

Proof of $Inv 1' \land Inv 2' \land Compat(0, 0, \text{false, false, } h_1, h_2) \implies Inter 3$ . For $ftok_2 \neq 0$, $X_1$ must read the non-zero value written to $token_2$ at $T_2$ and similarly for $stok_3 \neq 0$, $W_2$ must read the non-zero value written to $token_1$ at $T_1$. Let $X_1$ read from $T_2$ of iteration $k_2$ and $W_2$ reads from $T_1$ of iteration $k_1$. Following possibilities arise based on whether or not $k_1$ and $k_2$ are last iterations of Proc$_1$ and Proc$_2$.

$k_1$ is not the last iteration and $k_2$ is the last iteration In order to have $stok_1 \neq 0$, $W_2$ is placed after $T_1$ of $k_{1}^{th}$ iteration and before $P_1$ of $k_1^{th}$ iteration in the merged history. $Inv 2$ implies that $T_2 < W_2$ and therefore the value written to $token_2$ in the last iteration of Proc$_2$ at $T_2$ is some $v$ such that $v \neq 0$ and it flows to $W_2$. $Inv 1$ implies $P_1 < X_1$ and because Proc$_1$ does not write to $token_2$ hence $X_1$ also sees the value of $token_2$ as $v \neq 0$. $Inv 1$ further implies that $P_1 < R_1$ for $R_1$ of any iteration greater than $k_1$ and $R_1 < T_1$ such that $R_1$ reads the value of $token_2$ and writes back this value incremented by 1 to $token_1$ at $T_1$. Therefore $v+1$ is written to $token_1$ at $T_1$ in all iterations greater than $k_1$. Further, $T_1 < W_1$ implies that the same value $v+1$ is also visible at $W_1$. Therefore we get $ftok_1 > ftok_2$ implying Inter3 trivially holds.

$k_1$ is the last iteration and $k_2$ is not the last iteration In order to have $ftok_2 \neq 0$, $X_1$ is placed after $T_2$ of $k_2$ and before $P_2$ of $k_3$ in the merged history. $Inv 1$ implies $T_1 < X_1$ and therefore the value written to $token_1$ in the last iteration of Proc$_1$ at $T_1$ is some $v$ such that $v \neq 0$ and it flows to $X_1$. $Inv 2$ implies $P_2 < R_2$ for $R_2$ of any iteration greater than $k_2$ and $R_2 < T_2$ such that $R_2$ reads the value of $token_1$ and writes back this value incremented by 1 to $token_2$ at $T_2$. Therefore $v+1$ is written to $token_2$ at $T_2$ in all iteration greater than $k_2$. Also $Inv 2$ implies $T_2 < X_2$ which gives $stok_2$ equal to $v+1$. Further $Inv 2$ also implies $P_2 < W_2$ hence the same value $v$ of $token_1$ which is visible at $P_2$ is assigned to $stok_1$. This results in $stok_1 < stok_2$ and hence proved.

Both $k_1$ and $k_2$ are last iterations : In this case $X_1$ is placed after $T_2$ of the last iteration and because $T_2 < X_2$ hence $ftok_2 = stok_2$. Similarly, if $W_2$ is placed after $T_1$ of the last iteration then from $T_1 < W_1$ in $Inv 1$ we have $stok_1 = stok_2$.

Therefore $ftok_1 \leq ftok_2 \implies stok_1 \leq stok_2$ hence proved.

Neither of $k_1$ and $k_2$ are last iterations : We show that if $ftok_2$ reads the value of token$_2$ from $k_{2}^{th}$ iteration of Proc$_2$ then it is not possible for stok$_1$ to read the value of token$_1$ from the iteration $k_1$ of Proc$_1$ which is not the last iteration. Let $X_1$ is placed after $T_2$ and before $P_2$ of $k_2^{th}$ iteration. $T_1 < X_1$ implies that the value of token$_1$ visible at $X_1$ is from the last iteration of $T_1$ which also becomes visible at $P_2$ of $k_2^{th}$ iteration because of the placement of $X_1$. Further, $P_2 < W_2$ implies that $W_2$ sees the same value of token$_1$ written by the last iteration of Proc$_1$ at $T_1$ which
is not $k_1$. Similar argument follows for $W_2$ as well. Hence no compatible
merged history exists for this case.
Finally we collect the sufficient orderings used to prove $Inter3$.
- $T_1 \prec X_1$, $T_1 \prec W_1$, $P_1 \prec X_1$, $P_1 \prec R_1$ and $R_1 \prec T_1$ for $Proc_1$.
- $T_2 \prec X_2$, $T_2 \prec W_2$, $P_2 \prec W_2$, $P_2 \prec R_2$ and $R_2 \prec T_2$ for $Proc_2$.
In $P_1 \prec R_1$ and $P_2 \prec R_2$, $R_k$ comes from the iteration later than that of $P_i$.

Lamport’s bakery algorithm under PSO memory model
PSO memory model allows write instructions in a process to be reordered with future
write and read instructions operating on different addresses. We have following sufficient instruction orderings needed to prove $Inter$ and hence mutual exclusion.
- $T_1 \prec V_1$, $T_1 \prec W_1$, $T_1 \prec X_1$, $P_1 \prec X_1$, $V_1 \prec X_1$, $Q_1 \prec R_1$, $R_1 \prec T_1$
- $T_2 \prec V_2$, $T_2 \prec X_2$, $T_2 \prec W_2$, $P_2 \prec W_2$, $V_2 \prec X_2$, $Q_2 \prec R_2$, $R_2 \prec T_2$
- $P_1 \prec R_1$ if $P_1$ is from iteration $k$ then $R_1$ is from iteration greater than $k$.
- $P_2 \prec R_2$ if $P_2$ is from iteration $k$ then $R_2$ is from iteration greater than $k$.
The PSO memory model preserves the ordering of any two instructions which are
data or control dependent. Therefore $T_1 \prec W_1$, $T_2 \prec X_2$, $R_1 \prec T_1$ and
$R_2 \prec T_2$ are also satisfied by PSO. Further, PSO also preserves the order of
two read instructions, i.e. $V_1 \prec X_1$ and $V_2 \prec X_2$. A fence between $T_1$ and
$V_1$ also satisfies the ordering $T_1 \prec X_1$ and symmetrically a fence between
$T_2$ and $V_2$ also satisfies the ordering $T_2 \prec W_2$. Further, a fence between $Q_1$
and $R_1$ also orders $P_1$ of $k^{th}$ iteration before $R_1$ of $\geq k + 1^{th}$ iterations and
$X_1$. A fence between $Q_2$ and $R_2$ also orders $P_2$ of $k^{th}$ iteration before $R_2$ of
$\geq k + 1^{th}$ iteration and $W_2$. Therefore we only need two fence instructions
per process, i.e. between $Q_1$ and $R_1$ and between $T_1$ and $V_1$ in $Proc_1$ and
symmetrically in $Proc_2$.

6.2 Example: Simpson’s 4 slot algorithm [Sim90]
This is a wait-free algorithm for concurrent access of a location from a single
reader and a single writer processes. This location is simulated using a $2 \times 2$ array
variable slot. If the reader is reading the data and the writer wants to write new
data at the same time then instead of waiting for the reader to complete, the
writer writes to a different index of slot and indicates the reader to read from this
location in the subsequent read. Boolean variables reading and latest represent
two indices (false as 0 and true as 1) to denote the row and column indices of slot
and index variable. Variable index is a two element boolean array such that for
any $s \in \{true, false\}$, slot[$s$][index[$s$]] has the latest data written by the writer
in the row slot[$s$]. Variables fwp and fwindex are local to the writer process.
Variables srp and srindex are local to the reader process. The algorithm with inline
assertions on history and local variables are shown in Figure 10. In order to
simulate different invocations of the writer and the reader the respective program
is enclosed within the loop. We are interested in proving following two properties
of this algorithm.
**Interference Freedom** We want to show that at the entry point of critical sections (for the writer and the reader) $fwp \neq srp$, i.e. the reader and the writer use different rows of the slot variable, and $fwp = srp \Rightarrow fwindex \neq srindex$, i.e. if both use the same row of the slot variable then they read from and write to different column of that row. Therefore, we want to prove

\[
Inv1'(h_1) \land Inv2'(h_2) \land \text{Compat}(false, false, \{0\}, \{false\}, h_1, h_2) \implies fwp \neq srp \lor fwp = srp \Rightarrow fwindex \neq srindex
\]

Where $Inv1'$ and $Inv2'$ are the assertions just before the program point where writer is going to write the data and reader is going to read the data.

**Proof.** In any compatible merged history $h$ of $h_1$ and $h_2$ the placement of the last write ($\text{reading}, ph_1'$) of $h_2$ has two choices with respect to the last read ($\text{reading}, ph_1$) of $h_1$.

- *Last write* ($\text{reading}, ph_1'$) of $h_2$ *is placed before the last read* ($\text{reading}, ph_1$) of $h_1$: As writer does not write to reading hence $\text{ph}_1 = ph_1'$, or $\neg fwp = srp$ (from assertions $fwp = \neg ph_1$ and $srp = ph_1'$) and therefore $fwp \neq srp$. Hence proved.

Fig. 10. Simpson’s 4 slot algorithm
Last write (?reading, ph') of h₂ is placed after the last read (?reading, ph₁) of h₁: This, along with the assumption fwp = srp implies ?ph₁ = ph'₁. Therefore in Inv₂', (?index[ph₁], ph₂) is same as (?index[?ph₁], ph₂). According to Inv₁' (?index[?ph₁], ph₂) is in h₁. We want to establish the relation between ph₂ and ph'₂.

Inv₁' implies that S₁ < (?reading, ph₁) hence there is no write to variable index beyond (?reading, ph₁) in the merged history. Hence the value at index[?ph₁] is same for both processes, i.e. ph₂ = ph'₂. From Inv₁' and Inv₂' we get \( f\text{index} = ?ph₂ \) and \( s\text{index} = ph₂ \) which implies \( f\text{index} \neq s\text{index} \). Hence proved.

From the above proof, we can see that the only ordering important in proving this property is \( S₁ < P₁ \) where \( P₁ \) is from later iteration than that of \( S₁ \). Now we prove another property of interest and find out the program orderings used in that proof.

**Consistent Reads** Second property of interest is related to the order of reads. It specifies that the values read by the reader form a stuttering sequence of values written by the writer, i.e. if the writer writes the sequence 1,2,3,4,5 in subsequent invocations of write then the reader cannot observe 1,4,3,5 as a sequence read. It must observe the sequence which preserve the order of writes and possibly interspersed with the repetition of the same data. First, we define some notations used in this section. Let MergedHist(Reader, Writer) be the set of all compatible merged histories consisting of \( R \) invocations of the reader process and \( W \) invocations of the writer process. Let Reader(n) and Writer(n) be the \( n^{th} \) invocations of the reader and the writer respectively. Let \( D(k) \) be the data written by the writer in the \( k^{th} \) invocation. Let \( D(w) \) be a sequence \( D₁, D₂, \ldots, Dₙ \) of values written by \( w \) consecutive iterations of the writer. For \( s \in \{\text{true, false}\} \), \( \overline{s} \) denote the negation of \( s \). Let \( \text{elem}_{\text{Reader}} \) be the element \( \text{elem} \in \{(?\),(!\)\} \) in the merged history from \( r^{th} \) invocation of the Reader. Similarly \( \text{elem}_{\text{Writer}} \) denote the same for for the \( w^{th} \) iteration of the Writer.

**Stuttering sequence** Let \( S_{r,w} \) be a sequence of length \( r \), constructed from the elements of \( D(w) \). \( S_{r,w} \) is a stuttering sequence of \( D(w) \) if for any index \( i \) of the sequence \( S_{r,w} \) such the \( S_{r,w}[i - 1] = Dₖ₁ \) and \( S_{r,w}[i] = Dₖ₂ \) then \( k₁ ≤ k₂ \).

**Some interesting properties of the Reader and the Writer processes**

Only the writer writes to \( \text{latest} \) and reads from \( \text{reading} \) variable. Further, only the reader writes to \( \text{reading} \) and reads from \( \text{latest} \). Also, the value written to \( \text{reading} \) by the reader in any invocation is same as the value read from \( \text{latest} \). The value written to \( \text{latest} \) by the writer in any invocation is negation of the value that it reads from \( \text{reading} \).

Following lemma characterizes the sequence of values written to \( \text{reading} \) in a segment of the merged history. This characterization is then used in the proof of Lemma 2.
Lemma 1. For all \( R, W, h \in \text{MergedHist}(\text{Reader}^R, \text{Writer}^W) \), \( r \leq R, w \leq W, s \in \{\text{true, false}\} \), if \( \text{Reader}(r) \) reads the value of latest written by \( \text{Writer}(w) \) as \( s \) then in the sequence of values written to variable reading between \( P_1 \) of \( \text{Writer}(w) \) and \( P_2 \) of \( \text{Reader}(r) \), \( s \) is never followed by \( \overline{s} \).

\[ A'' : (\text{latest}, \overline{s})_{\text{Writer}^{n'}} \quad B' : (\text{latest}, \overline{s})_{\text{Reader}^r} \quad B' : (\text{reading}, \overline{s})_{\text{Reader}^r} \]

\[ A' : (\text{reading}, \overline{s})_{\text{Writer}^{n+1}} \quad A : (\text{latest}, s)_{\text{Writer}^{n+1}} \]

\[ B : (\text{latest}, s)_{\text{Reader}^r} \]

Fig. 11. Merged history for Lemma 1

Proof. We prove it using induction on the iteration number of the writer process.

**Base case, \( w = 0 \):** If \( \text{Reader}(r) \) reads the initial value of latest (in 0th iteration of the writer process), say \( s \), then all the invocations of the reader before \( \text{Reader}(r) \) also see this initial value of latest as \( s \) and therefore the sequence is made of only this value. Hence the base case satisfies this property.

**Induction Hypothesis, \( w \leq n \):** For all \( w \leq n \), if \( \text{Reader}(r) \) reads the value of latest as \( s \), written by \( \text{Writer}(w) \) then the sequence of values written to reading between \( P_1 \) of \( \text{Writer}(w) \) and \( P_2 \) of \( \text{Reader}(r) \) satisfies this property.

**Induction Step, \( w = n + 1 \):** Consider the merged history of Figure 11 where \( \text{Reader}(r) \) reads the value of latest (denoted by \( B \)) from \( \text{Writer}(n + 1) \) (denoted by \( A \)). We want to prove this property for the sequence of values written to reading between \( \text{A}' \) and \( B \) where \( \text{A}' \) denotes \( P_1 \) of \( \text{Writer}(n + 1) \) in Figure 11. \( P_1 \prec T_1 \) implies that \( \text{A}' \) appears before \( A \) in the merged history. Let \( \text{Writer}(n + 1) \) read the value of reading from \( \text{Reader}(r') \) (denoted by \( B' \)). \( P_2 \prec Q_2 \) implies that \( \text{P}_2 \) of \( \text{Reader}(r') \) appears before \( B' \) in the merged history. Let \( \text{Reader}(r') \) at \( P_2 \) read the value of latest from \( \text{Writer}(n')(A'') \). It is clear that \( n' < n + 1 \) hence from the induction hypothesis we know that in the sequence of values written to reading between \( A'' \) and \( B'' \), \( \overline{s} \) is never followed by \( s \). We use this knowledge to characterize the values written to latest between \( B'' \) and \( \text{A}' \). From the writer process we know that the
value written to latest is negation of the value read from reading variable. Therefore in the sequence of values written to latest between B′ and A′, s is never followed by T. Further, we also know that the reader process writes the same value to reading as is read from the variable latest. Therefore in the sequence of values written to reading between A′ and B, s is never followed by T. Hence proved.

For any reader we establish the relation between its read of variable latest and the read of data. More formally we want to say that,

**Lemma 2.** For all R, W, h ∈ MergedHist(ReaderR, WriterW), r ≤ R, w ≤ W, s ∈ {true, false}, if Reader(r) reads the value of latest as s written by Writer(w) and reads the value of index[s] written by Writer(w′) then w′ ≥ w and Reader(r) reads the data Dw′ and all the invocations of the writer from w to w′ write the data in the row s of the slot variable.

**Proof.** Let us assume that Reader(r) reads the value of latest as s written by Writer(w) which means that P2 of Reader(r) appears after T1 of Writer(w) and no write to latest appears in between these two points in the merged history. As Writer(w) also writes to index[s] and S1 ≺ T1 and P2 ≺ R2, hence Reader(r) reads the value of index[s] either from Writer(w) or from Writer(w′) such that w′ ≥ w. We prove the following two properties,

- **Reader(r) reads the data written by Writer(w′) in slot[s][k] where k is the value written to index[s] by Writer(w′)**

  **Proof:** If Reader(r) reads index[s] as k ∈ {true, false} from Writer(w′) then R1 ≺ S1 implies that Writer(w′) has also written the data in slot[s][k]. We want to show that no subsequent invocation of the writer writes in slot[s][k] before S2 of Reader(r). From the assumption that R2 of Reader(r) reads the value of index[s] from Writer(w′) implies that there is no write to index[s] between S1 of Writer(w′) and R2 of Reader(r). S1 ≺ P1, where P1 is from later iterations than that of S1, in Inv1′ implies that the writer can be invoked at most once after the iteration w′ and before R2 of Reader(r). If invoked more than once it results in the write of index[s] to appear after S1 of Writer(w′) and before R2 of Reader(r), contradicting our assumption. Further, if the next invocation after w′ happens then it writes the data to column k of row s in slot variable still satisfying the property. If any further invocation of the writer happens after R2 and before S2 of Reader(r) then because of Q2 ≺ R2 it observes the value of reading as s and therefore writes the data to T row of the slot variable. Inv2′ implies that S2 of Reader(r) ≺ Q2 of subsequent invocations of the reader and therefore no write to reading exists between R2 and S2 of Reader(r). This results in observing the same value of reading, s, and subsequently writing the data to row T by all invocations of the writer between R2 and S2 of Reader(r).

- **All the invocations of the writer from w to w′ write the data in the row s of the slot variable**

  **Proof:** We prove an alternate equivalent property; All invocations of
the writer from \( w \) to \( w' \) read the value of \( \text{reading} \) as \( \mathbf{s} \). This follows from the property that the writer writes in that row of the slot variable that is obtained by negating the value of the variable \( \text{reading} \). From the assumption, \( \text{Writer}(w') \) writes to \( \text{index}[s] \) and therefore it also reads the value of \( \text{reading} \) as \( \mathbf{s} \) and the same holds for \( \text{Writer}(w) \) as well. We need to show that this property holds for the rest of the invocations in between \( w \) and \( w' \). We prove it by contradiction by assuming that there exists a \( w'' \) such that \( w < w'' < w' \) and \( P_1 \) of \( \text{Writer}(w'') \) reads the value of \( \text{reading} \) as \( s \). It is clear that \( P_1 \) of \( \text{Writer}(w'') \) appears after \( P_1 \) of \( \text{Writer}(w) \) and before \( P_1 \) of \( \text{Writer}(w') \). From the earlier argument we know that the value of \( \text{reading} \) visible at \( P_1 \) of \( \text{Writer}(w) \) is \( \mathbf{s} \). Hence, \( P_1 \) of \( \text{Writer}(w'') \) cannot read from the value of \( \text{reading} \) visible at \( P_1 \) of \( \text{Writer}(w) \). Therefore, it must read the value of \( \text{reading} \) as \( s \) from the sequence of values written to \( \text{reading} \) between \( P_1 \) of \( \text{Writer}(w) \) and \( P_2 \) of \( \text{Reader}(r) \). Following Lemma 1, \( s \) can never be followed by \( \mathbf{s} \) in the sequence of values written to \( \text{reading} \) in between \( P_1 \) of \( \text{Writer}(w) \) and \( P_2 \) of \( \text{Reader}(r) \). It further implies that \( P_1 \) of \( \text{Writer}(w') \) cannot read the value of \( \text{reading} \) as \( \mathbf{s} \), a contradiction to our assumption. Therefore all invocations of the writer from \( w \) to \( w' \) read the value of \( \text{reading} \) as \( \mathbf{s} \). Hence proved.

Now we use Lemma 2 to prove the following theorem.

**Theorem 1 (Stuttering Sequence).** \( \text{Stutter}(n) \overset{\text{def}}{=} \text{For all } n, w \in \mathbb{N}, h \in \text{MergedHist(Reader}^n, \text{Writer}^w) \text{ the sequence } S_{n,w}, \text{ constructed from the elements of } D(w), \text{ is a stuttering sequence of } D(w). \)

**Proof.**

**Base case,** \( n = 0 \): With no history corresponding to the Reader in the merged history, the empty sequence trivially forms a stuttering sequence.

**Induction Hypothesis:** For all \( r \leq n \) \( \text{Stutter}(r) \) holds true.

**Induction Step,** \( r = n + 1 \): Let \( \text{Reader}(n) \) read the value of \( \text{latest} \) from \( \text{Writer}(w_a) \). From Lemma 2, we know that \( \text{Reader}(n) \) reads the value \( D^{w'} \) such that \( w_a \leq w' \leq w \) and all the invocations of the writer from \( w_a \) to \( w' \) write the data in the same row \( s \) of the slot variable. Further, the value of \( \text{index}[s] \) visible at \( R_2 \) of \( \text{Reader}(n) \) is from \( S_1 \) of \( \text{Writer}(w') \). We know that the write to \( \text{latest} \) from consecutive iterations are totally ordered and the same holds for consecutive reads from \( \text{latest} \) as well. Therefore, \( \text{Reader}(n + 1) \) reads the value of \( \text{latest} \) from some \( \text{Writer}(w_b) \) such that \( w_b \geq w_a \). There are two possibilities based on whether this value is \( s \) or \( \mathbf{s} \).

- \( \text{Reader}(n + 1) \) reads the value of \( \text{latest} \) as \( s \) from the \( \text{Writer}(w_b) \) such that \( w_b \geq w_a \): From our assumption we know that \( \text{Reader}(n) \) sees the value of \( \text{index}[s] \) from \( \text{Writer}(w') \) therefore the value of \( \text{index}[s] \) visible at \( R_2 \) of \( \text{Reader}(n + 1) \) will be from some \( w'' \) such that \( w \geq w'' \geq w' \). Following Lemma 2, the data read by \( \text{Reader}(n + 1) \) from \( \text{slot}[s][\text{index}[s]] \) will be from \( \text{Writer}(w'') \) and \( w'' \geq w' \) implies that the resulting sequence \( S_{n+1,w} \) is still a stuttering sequence.
Reader\((n+1)\) reads the value of \textit{latest} as \(\pi\) from Writer\((w_b)\) such that \(w_b \geq w_a\). From Lemma 2, we know that all the invocations of the writer from \(w_a\) to \(w'\) write the data in the same row \(s\) of slot and because Writer\((w_b)\) writes the data in row \(\pi\) of slot therefore \(w_b > w'\). Following Lemma 2, Reader\((n+1)\) reads the data from some Writer\((w''\) such that \(w'' \geq w_b\). Combining these two we get \(w'' > w'\) such that Reader\((n)\) reads \(D^{w'}\) and Reader\((n+1)\) reads \(D^{w''}\). Therefore the resulting sequence \(S_{n+1,s}\) is still a stuttering sequence.

\textit{Simpson’s 4 slot algorithm under PSO memory model} Following per-thread instruction orderings are used in the proofs of the interference freedom and the consistent reads properties.

\begin{itemize}
  \item \(P_1 \prec T_1\) (from the proof of Lemma 1), \(S_1 \prec T_1\), \(R_1 \prec S_1\)
  \item \(P_2 \prec Q_2\) (from the proof of Lemma 1), \(P_2 \prec R_2\), \(Q_2 \prec R_2\)
  \item \(S_1 \prec P_1\), where \(P_1\) is from iteration later than that of \(S_1\)
  \item \(S_2 \prec Q_2\), where \(Q_2\) is from iteration later than that of \(S_2\).
\end{itemize}

Out of all these orderings, \(P_1 \prec T_1\), \(P_2 \prec Q_2\) and \(P_2 \prec R_2\) are data dependent orders which are respected by the PSO memory model. \(S_2 \prec Q_2\) where \(Q_2\) is from iteration later than that of \(S_2\) is also respected by PSO memory model because \(S_2\) corresponds to read and \(Q_2\) corresponds to write instruction. Therefore, we have to enforce only \(R_1 \prec S_1\), \(S_1 \prec T_1\), \(Q_2 \prec R_2\) and \(S_1 \prec P_1\) where \(P_1\) is from iteration later than that of \(S_1\). Following the semantics of \textit{fence} it is sufficient to put two fence instructions in the writer; one between \(R_1\) and \(S_1\) and another between \(S_1\) and \(T_1\). Further, we need one fence instruction between \(Q_2\) and \(R_2\) in the reader as well.

\section{Conclusion and Future Work}

In this paper we proved Simpson’s 4 slot algorithm correct under the SC memory model with respect to the interference freedom and the consistent reads properties. Based on these proofs, we identified the locations of the \textit{fence} instructions needed to satisfy these two properties under the PSO memory model. As a direction for future work we still have to explore the use of this approach for advanced memory models which support non-atomic writes (POWER/ARM). This paper introduces the predicates over history variable only to carry out proofs conveniently. No formal treatment is given to them and this should be addressed with high priority. Even in presence of predicates over history variables, the difficulty in carrying out these proofs without any tool support is evident from the proof of Simpson’s 4 slot algorithm. We plan to use proof assistants for this in near future. In all the examples we considered here, a program is always executed by only one thread. This restriction needs to be addressed in order to handle concurrent data structures where many threads can execute the same method of an object.
References

[AAC+12a] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Automatic fence insertion in integer programs via predicate abstraction. In SAS, pages 164–180, 2012.

[AAC+12b] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, Ahmed Rezine, and Ahmed Rezine. Counter-example guided fence insertion under tso. In TACAS, pages 204–219, 2012.

[ABBM10] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. SIGPLAN Not., 45(1):7–18, January 2010.

[ABBM12] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. What’s decidable about weak memory models? In ESOP, pages 26–46, 2012.

[BD12] Richard Bornat and Mike Dodds. Abducing memory barriers. Draft, April, 2012.

[Bor12] Richard Bornat. Barrier logic: a program logic for concurrency on powerpc. 2012.

[HR07] Thuan Quang Huynh and Abhik Roychoudhury. Memory model sensitive bytecode verification. Form. Methods Syst. Des., 31(3):281–305, December 2007.

[Jon83] C. B. Jones. Specification and design of (parallel) programs. In Proceedings of IFIP’83, pages 321–332. North-Holland, 1983.

[KVY11] Michael Kuperstein, Martin Vechev, and Eran Yahav. Partial-coherence abstractions for relaxed memory models. In Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, PLDI ’11, pages 187–198, New York, NY, USA, 2011. ACM.

[Lam74] Leslie Lamport. A new solution of dijkstra’s concurrent programming problem. Commun. ACM, 17(8):453–455, August 1974.

[LW11] Alexander Linden and Pierre Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In Proceedings of the 18th international SPIN conference on Model checking software, pages 144–160, Berlin, Heidelberg, 2011. Springer-Verlag.

[O’H07] P.W. O’Hearn. Resources, concurrency, and local reasoning. Theoretical Computer Science, 375(1):271–307, 2007.

[Rid10] Tom Ridge. A rely-guarantee proof system for x86-tso. In Proceedings of the Third international conference on Verified software: theories, tools, experiments, VSTTE’10, pages 55–70, Berlin, Heidelberg, 2010. Springer-Verlag.

[Sim90] H.R. Simpson. Four-slot fully asynchronous communication mechanism. Computers and Digital Techniques, IEE Proceedings E, 137(1):17 – 30, jan 1990.

[Sou84] N. Soundararajan. A proof technique for parallel programs. Theoretical Computer Science, 31(1–2):13 – 29, 1984.

[VP07] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In CONCUR 2007, pages 256–271. Springer, 2007.