A Verification-Based Approach to Memory Fence Insertion in PSO Memory Systems*

Alexander Linden and Pierre Wolper

Institut Montefiore, B28
Université de Liège
B-4000 Liège, Belgium
{linden,pw}@montefiore.ulg.ac.be

Abstract. This paper addresses the problem of verifying and correcting programs when they are moved from a sequential consistency execution environment to a relaxed memory context. Specifically, it considers the PSO (Partial Store Order) memory model, which corresponds to the use of a store buffer for each shared variable and each process. We also will consider, as an intermediate step, the TSO (Total Store Order) memory model, which corresponds to the use of one store buffer per process.
The proposed approach extends a previously developed verification tool that uses finite automata to symbolically represent the possible contents of the store buffers. Its starting point is a program that is correct for the usual Sequential Consistency (SC) memory model, but that might be incorrect under PSO with respect to safety properties. This program is then first analyzed and corrected for the TSO memory model, and then this TSO-safe program is analyzed and corrected under PSO, producing a PSO-safe program. To obtain a TSO-safe program, only store-load fences (TSO only allows store-load relaxations) are introduced into the program. Finaly, to produce a PSO-safe program, only store-store fences (PSO additionally allows store-store relaxations) are introduced.
An advantage of our technique is that the underlying symbolic verification tool makes a full exploration of program behaviors possible even for cyclic programs, which makes our approach broadly applicable. The method has been tested with an experimental implementation and can effectively handle a series of classical examples.

1 Introduction

Modern multiprocessor architectures optimize accesses to shared memory and, doing so, do not implement the traditional *Sequential Consistency* (*SC*) memory model [1], in which all accesses to the shared memory are immediately visible globally. The exact behavior of these processors with respect to memory accesses is rather complex and is usually only described by a set of typical behaviors in vendor documentation. Nevertheless, formal models that cover the behavior

* This work is supported by the grant 2.4545.11 of the Belgian Fund for Scientific Research (F.R.S.-FNRS).

of many existing processors have been defined. These are usually referred to as *relaxed memory models*, among the most common being *Total Store Order* (*TSO*) and *Partial Store Order* (*PSO*), both defined in [2, 3]. Writing correct code under these models is quite challenging given that they allow even more executions than the traditional SC model. This has motivated work on verifying code under these memory models, as well as on techniques for preserving the correctness of code when it is moved from the SC model to a relaxed memory model. This is done by introducing forced memory synchronizations known as *fences*. However, using fences means forgoing the benefits of the hardware optimizations that lead to relaxed memory models, so the issue is to minimize the number of inserted fences.

In earlier work, [4, 5], we proposed a technique that models TSO using store buffers and uses finite automata to represent the potentially infinite set of possible contents of these buffers. This representation coupled with acceleration techniques similar to those proposed in [6], as well as with the *persistent-set* and *sleep-sets* partial-order reduction techniques [7], allows a full exploration of the state space of programs, including for cyclic programs. In this earlier work both the problem of verifying a program under TSO and of inserting fences to preserve the correctness of a program being moved from SC to TSO are addressed.

This paper focuses on porting a program verified under SC to PSO, while preserving its safety properties. The approach is based on the verification techniques and tool already presented in [4, 5] for TSO. The first contribution of this paper is to extend these techniques and the tool to PSO. One challenge that had to be solved for doing this is that in PSO, a single process writes to several buffers, one for each variable. Thus when dealing with the repetition of cycles, it seems necessary to synchronize the writes to different buffers, hence taking us beyond what can be represented with finite automata. Fortunately, as we will establish in this paper, this synchronization can safely be ignored.

The second contribution of the paper is a method for safely porting programs from SC to PSO. It starts by analyzing and correcting the program under TSO, inserting the necessary memory fences [5]. The fences inserted here are *mfences*, which is what is required to avoid the store-load relaxations possible in TSO. The second step is to move a program safe under TSO to PSO. Here, the approach is similar to the first step, but we only use *sfences*, which are weaker, but sufficient for avoiding the store-store relaxations possible under PSO. We present experimental results that show that the approach is quite effective, can efficiently handle a number of meaningful examples, and compares favorably to other methods proposed for the same problem.

2 Concurrent Programs and Memory Models

We consider a very simple model of concurrent programs in which a fixed set of finite-state processes are interacting through a shared memory. Such a concurrent system is thus defined by a finite set of processes $\mathcal{P} = \{p_1, \ldots, p_n\}$ and a finite set of memory locations $\mathcal{M} = \{m_1, \ldots, m_k\}$, the initial content of the shared

memory being defined by a function $\mathcal{I} : \mathcal{M} \to \mathcal{D}$, $\mathcal{D}$ being the domain of memory values.

The definition of each process p_i includes a finite set of control locations $\mathcal{L}(p_i)$, an initial control location $\ell_0(p_i) \in \mathcal{L}(p_i)$ and a set of transitions $\mathcal{T}$ labeled by operations taken from a set $\mathcal{O}$. The transitions of a process p_i are thus elements of $\mathcal{L}(p_i) \times \mathcal{O} \times \mathcal{L}(p_i)$, also written as $\ell \xrightarrow{op} \ell'$, where both $\ell, \ell' \in \mathcal{L}(p_i)$.

The set $\mathcal{O}$ of operations contains the two following memory operations:

- $store(p, m, v)$, the meaning of which is that process p stores the value $v \in \mathcal{D}$ to the memory location m,
- $load(p, m, v)$, the meaning of which is that process p loads the value stored in memory location m and checks if that value is equal to v. The operation is possible only if the values are equal, otherwise it does not go through and execution is blocked.

Under the SC memory model, the semantics of such a concurrent program is the one in which the possible behaviors are all the interleavings of the operations executed by the different processes, and in which the store operations become immediately visible to all processes.

In TSO, each process executing a store operation can directly load the value saved by this store operation, but other processes cannot always immediately see that value and might read an older value stored in shared memory. This is known as the fact that TSO allows the *store-load* relaxation. PSO also allows such *store-load* relaxations to happen but, additionally, stores accessing different shared memory locations can be reordered within the same process, which is known as the *store-store* relaxation. Thus, the possible SC-executions are included in the set of TSO-executions, which are themselves included in the set of PSO-executions.

The formal definitions of the memory models use the concepts of *program order* and *memory order* [2, 8]. Program order ($<_p$) is a partial order in which the instructions of each process are ordered as executed, but instructions of different processes are not ordered with respect to each other. Memory order ($<_m$) is a total order on the memory operations, which is fictitious but characterizes what happens during relaxed executions.

Let l or l^i denote any load operation, s any store operation, l_a a load operation on location a, and s_a or s_a^i store operations on location a. Furthermore, let $val(l)$ be the value returned by the load operation l.

Using these notions, a formal definition of PSO can be given (for the definitions of SC and TSO, see [2, 8] or [5]).

A PSO execution is one for which there exists a memory order satisfying the following constraints for each process p:

1. $\forall l^1, l^2 : l^1 <_p l^2 \Rightarrow l^1 <_m l^2$
2. $\forall l, s : l <_p s \Rightarrow l <_m s$
3. $\forall s_a^1, s_a^2 : s_a^1 <_p s_a^2 \Rightarrow s_a^1 <_m s_a^2$
4. $val(l_a) = val(\max_{<_m}\{s_a \mid s_a <_m l_a \vee s_a <_p l_a\})$. If there is no such a s_a, $val(l_a)$ is the initial value of the corresponding memory location.

The first three rules specify that the memory order has to be compatible with the program order, except that a store can globally be postponed after a later load or a later store accessing a different variable of the same process. The last rule specifies that the value retrieved by a load is the one of the most recent store in memory order that precedes the load in memory order or in program order, the latter ensuring that a process can see the last value it has stored. If there is no such store, the initial value of that memory location is loaded.

This axiomatic definitions of PSO gives insight, but the equivalent operational model is much more useful for applying explicit state-space exploration techniques. This operational model is described in Fig.1. Stores from each process are buffered, a separate buffer being used by each process for each shared memory location. A store only takes effect when it is transferred from a buffer to the shared memory, which is called a *commit*. This can be seen as the moment when it is entered into the memory order. A commit operation is an internal system operation, which is assumed to be executed nondeterministically for each buffer and each process. This model (using buffers and commits) ensures that stores by the same process accessing the same locations cannot be reordered, while those accessing different locations can. When a load is executed by a process, it will read the most recent value out of its own store buffer for this variable if there exists at least one buffered store to that variable, otherwise the load reads the value out of the shared memory. This means that loads can be reordered with earlier stores of the same process, while they always read the most recent values either from a buffer or the main memory.

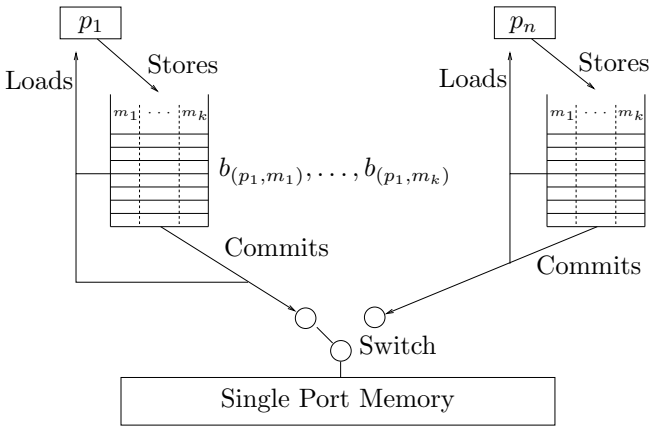

Fig. 1. Operational definition of PSO [2, 3]

To match what is available in actual processors, in particular Intel's x86 processors, extensions have to be made to TSO and PSO [9, 10]. The first extension is adding a new component, the lock, which is used to grant processes exclusive

access to the shared memory. The second extension consists of operations called memory fences, which constrain how stores are committed to main memory. In TSO, only one type of fence is available, the *mfence*. An *mfence* operation blocks the executing process until every earlier executed store operation of that process has been committed to the shared memory. In PSO, a second type of fence is also available, the *sfence*. When an *sfence* occurs in a process, it forces every store preceding the *sfence* to be committed to memory before every store that occurs after the *sfence*. An *sfence* does not block the process executing it, but of course restricts the execution of commit operations. When comparing sfences and mfences, it is clear that the effect of an mfence is stronger than the effect of an sfence. The mfence disables all relaxations between operations before and after the mfence, whereas the sfence only disables the *store-store* relaxations.

To formally define the operational model of PSO, we first add a set

$$\mathcal{B} = \{b_{(p_1,m_1)}, \ldots, b_{(p_1,m_k)}, b_{(p_2,m_1)}, \ldots, b_{(p_n,m_k)}\}$$

of buffers to the system, each process having one store buffer per variable[1]. Secondly, we add a global lock L component whose value can be a process $p \in \mathcal{P}$ when p holds the lock, or undefined ($\perp$) when the lock is not held by a process. A global state of the system becomes the composition of the content of the memory, the value of the global lock, and, for each process p, a control location as well as the content of its store buffers $[b_{(p,m_1)}, \ldots, b_{(p,m_k)}]$. The content of a buffer is a sequence of elements that are either (1) triplets (m, v, t) where $m \in \mathcal{M}$, $v \in \mathcal{D}$ and $t \in \mathcal{T}$, representing a store operation and identifying the transition where it was executed, or (2) a special symbol $\star^t$ representing an *sfence(p)* transition t. These semantics are very similar to those that were given for TSO in [5], and thus we will focus only on the operations that are specific to, or different in, PSO: *sfence* and *commit*.

sfence operation : *sfence*(p):

$$\forall m \in \mathcal{M} : [b_{(p,m)}] \leftarrow [b_{(p,m)}]\ \star^t,$$

where t is the transition corresponding to the current *sfence* operation.

commit operation : *commit*(p, m):

If ($[L] \neq \perp$ and $[L] \neq p$), where L is the lock, then *commit*(p, m) cannot be executed;
otherwise, let $[b_{(p,m)}] = (m, v_1, t_1)(m, v_2, t_2) \ldots (m, v_f, t_f)$ (the first element to commit is not an *sfence*). Then, if $[b_{(p,m)}] \neq \varepsilon$, the result of the commit operation is $[b_{(p,m)}] \leftarrow (m, v_2, t_2) \ldots (m, v_f, t_f)$ and $[m] \leftarrow v_1$, or, if $[b_{(p,m)}] = \varepsilon$, the commit operation has no effect. If $[b_{(p,m)}] = \star^t(m, v_1, t_1) \ldots (m, v_f, t_f)$, i.e. the buffer content starts with the symbol representing the *sfence*(p) operation of transition t, then *commit*(p, m) becomes a synchronized operation

[1] Note that we introduce the buffers per *process* rather than by *processor*. This approach is safe for verification since it allows more behaviors than a model in which some processes could share the same buffer. Furthermore, it is impossible to know which process will run on which processor when analyzing a program.

which requires all buffers of p to start with $\star^t$. If this is not the case, the commit cannot be executed. If all buffers start with $\star^t$, the commit operation can be executed, and simultaneously removes the element $\star^t$ from all buffers.

Note that *commit(p,m)* is not an operation that can appear in a program, but is assumed to be always enabled and nondeterministically interleaved with the actual program operations. Thus, when an *mfence(p)*, *unlock(p)* or the *sfence(p)* operation is blocked because the buffers of p are not all empty, or because not all buffers of p start with the same $\star^t$, the implicit execution of *commit(p,m)* operations makes it possible to empty the buffers of p or to reach $\star^t$ for all buffers of p, and enable the operation.

3 Representing Sets of Buffer Contents and State Space Exploration

Verifying a program under the TSO or PSO memory models can be done with a tool such as SPIN ([11]). However, this leads to two problems. First, one must bound the size of the buffers in order to keep the model finite-state. Second, the size of the state space quickly explodes as the size of the buffers grows.

These problems were addressed in [4], for TSO, as follows. To start with, rather than limiting buffers to a fixed size, finite automata, called buffer automata, are used to represent possibly infinite sets of buffer contents. Such buffer automata represent sets of unbounded buffer contents, those contained in the accepted language ($L(A)$) of the buffer automata (A). This allows unbounded buffer contents to be taken into account and, with the help of acceleration techniques similar to those of [12] and [6], to explore the full state space of programs, even if they include memory accesses, in particular memory writes, in cycles that can be infinitely repeated. The cycles that actually need to be, and can be, "accelerated" are those in which one particular process repeatedly writes to memory, thus potentially leading to an unbounded buffer content.

For PSO, the situation is similar, except that we need to handle not just one buffer per process, but a set of buffers, one for each variable and that we also need to handle *sfence operations*. The state-space exploration, including the use of partial-order techniques, as well as the detection of cycles is done exactly as for TSO, see [4]. What changes are the operations applied to the buffer automata to accelerate the cycles: rather than operating on a single automaton for each cycle, the one corresponding to the active process, we need to operate on multiple automata, one for each updated variable of the active process. The obvious way to do this is to filter from the cycle the operations corresponding to each variable and only consider these when dealing with the corresponding buffer automaton. This is straightforward to implement, but generates more buffer contents than can actually occur: the link between the number of times write operations are applied to different variables is lost! To make this clear, let us examine an example.

Consider the program given in Fig 2. It contains just one process with memory locations x, y and z all set to 0 initially. There will be a cycle detected after

the sequence of states $1 \to 2 \to 1 \to 2 \to 1$, and the content of the buffers for x, y and z will then be modified to be $((x,1,t_1)(x,1,t_1)^*;(y,1,t_2)(y,1,t_2)^*;\varepsilon)$. However, since the number of stores to x and y are the same, the accurate representation of the buffer contents after iterating the cycle would be $((x,1,t_1)(x,1,t_1)^n;(y,1,t_2)(y,1,t_2)^n;\varepsilon)$, and thus by considering the variables separately we have introduced buffer contents that cannot be generated by iterating the cycle. Fortunately, this is not a problem since committing several times the same memory write operation has no influence on the possible future behaviors of the program. More precisely, any program behavior that is possible from a global state with buffer contents $((x,1,t_1)(x,1,t_1)^{n_1};(y,1,t_2)(y,1,t_2)^{n_2};\varepsilon)$ with $n_1 \neq n_2$ is also possible from the corresponding global state with buffer contents $((x,1,t_1)(x,1,t_1)^{\max(n_1,n_2)};(y,1,t_2)(y,1,t_2)^{\max(n_1,n_2)};\varepsilon)$ by applying different numbers of commit operations to the variables x and y.

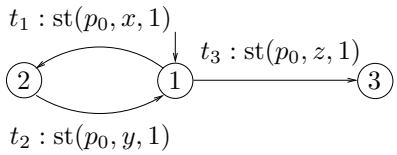

Fig. 2. A program with writes to different variables in a cycle

We now need to generalize the observation made in the previous example. To do this, we have to compare the executions that are possible if we compute the buffer contents resulting from the repeated execution of a cycle separately for each variable, or if we take into account the necessary *synchronization* of the operations performed on the different variables. We will refer to these as *synchronized* versus *unsynchronized* executions. For this we use the following concepts.

Definition 1. *Given a word w over an alphabet Σ and $L \subseteq \Sigma^+$, a word w' is a* ***L* stutter subword** *of w if w can be obtained from w' by, for one or more subwords u of w with $u \in L$, replacing u by a word in u^+.*

Example. The word *aabc* is a $\{b, c, bc\}$ stutter subword of *aabbbcc* and *aabcbcbc*.

Definition 2. *A sequence of operations that does not modify the store buffer in a way that affects the result of subsequent load operations is called* **load-preserving**.

We can then formalize the fact that repeating load-preserving sequences of commit operations has no real impact on an execution.

Lemma 1. *Le σ be an execution of a concurrent system and let LE be the set of load-preserving commit operation sequences appearing in σ. Then every LE stutter subword σ' of σ is also a valid execution of the system.*

Proof. This is a direct consequence of the fact that load-preserving sequences of commit operations are idempotent, i.e. applying them one or several times has no effect on the rest of the execution.

From this Lemma, it is easy to establish the property we need.

Theorem 1. *Computing the buffer automata of different variables independently only leads to valid executions.*

Proof. Indeed, the potentially incorrect executions that could be obtained by handling the buffers for different variables independently are those in which the number of stores to variables executed in the same cycle could be taken to be different. Notice that this will only have an effect on the execution when these stores are committed to memory and that committing the stores appearing on a cycle is load-preserving. Thus, such an unsynchronized execution will always be a LE stutter subword of a synchronized execution, where LE is the set of load-preserving commit sequences corresponding to cycles, and hence will be valid. Indeed, since we allow unbounded repetition of cycles, the synchronized execution can be taken to be the one in which the cycle is repeated a number of times greater than the largest number of times a store to any of the variables modified in the cycle is committed to memory.

After having introduced buffer automata representing sets of buffer contents rather than single buffer contents, one needs to redefine the operations on buffers to also apply to buffer automata. For the operations *store* and *mfence*, please refer to [4, 5].

load operation : $load(p, m, v)$:

The problem with a load operation applied to a buffer automaton is that it may succeed on some contents of the buffer represented by the automaton and fail on others. Thus, once a load was successfully applied to a buffer automata, we need to restrict the possible buffer contents to those on which the load operation succeeds (see [4]). But now that we are dealing with PSO, special care should be applied if $\star^t$ symbols are present. Indeed, if a $\star^t$ symbol is removed when modifying a buffer to take into account the fact a load has succeeded, the synchronization required by the *sfence* will no longer be possible, thus introducing a fictitious deadlock. If this occurs the buffers for the other variables of the process are also modified in order to remove the now spurious $\star^t$ symbols.

sfence operation : $sfence(p)$:

$$\forall m \in \mathcal{M} : L(A_{(p,m)}) \leftarrow L(A_{(p,m)})\ \star^t,$$

where t is the transition corresponding to the current *sfence* operation.

commit operation : $commit(p)$:

As for the load operation, the commit also may have an impact on the

possible buffer contents. How to restrict the buffer contents to those that match the current commit operation has been described in [4]. The related problem due to the sfences, as described above for the load operations, also occurs for the commit operation and is handled similarly.

4 From SC to TSO to PSO

We now turn to the problem of preserving the correctness of a program when it is moved from an SC to a PSO memory environment. By correctness, we mean preserving state (un)reachability properties. Note that this captures safety properties, since safety can always be reduced to state (un)reachability in an extended model.

An obvious way to make sure a program can safely be moved from SC to PSO is to force writes to be immediately committed to main memory by inserting an mfence after each store, thus precluding any process from moving with a nonempty store buffer. The obvious drawback of doing so is that any performance advantage linked to the use of store buffers in the implementation is lost.

However, it is not at all necessary to guarantee that the executions that can be seen under PSO are also possible under SC. We might rather just restrict the possible executions to those satisfying the desired safety property, i.e. only exclude those executions reaching states violating the safety property. Recall that the difference between SC, TSO and PSO can be summarized as follows: SC does not allow any relaxation, TSO allows the *store-load* relaxation, and PSO allows the *store-load* and the *store-store* relaxations. When needed, these relaxations can be avoided by placing adequate fences into the program.

In [5], we exploited this to maintain correctness of a program (wrt a safety property) when it was moved from SC to TSO. In the current approach, we want to go further and maintain correctness of a program when it is moved from SC to PSO. We will do this by first modifying the program to guarantee that it is still correct under TSO, and then further modify it so that it remains correct under PSO.

To avoid all relaxations, it is sufficient to place an *mfence* between all loads and any preceding store, as well as an *sfence* between stores accessing different variables. If this is the case, no relaxation will be possible, and all PSO executions will also be SC executions. As our approach proceeds in two steps, the first of which is described in [5], we now only need to describe how to avoid the *store-store* relaxations allowed in PSO, but not in TSO. Lemma 2 gives a sufficient condition for guaranteeing this..

Lemma 2. *Given a PSO execution, if in the program order of each process, an* sfence *is executed between every pair of successive stores accessing different memory locations, the memory order satisfies all the TSO constraints.*

Proof. The semantics of sfence operation can be formalized by introducing these operation in the memory order with the following constraints, where s_a represents a store operation accessing memory location a, and S represents an *sfence* operation:

1. $\forall s_a, S : s_a <_p S \Rightarrow s_a <_m S$
2. $\forall s_a, S : S <_p s_a \Rightarrow S <_m s_a$

In the conditions of the lemma, we have if $s_a <_p s_b$, there is an *sfence* S such that $s_a <_p S$ and $S <_p s_b$, and thus we have that $s_a <_m s_b$. It follows that the memory order thus satisfied all constraints of a TSO order. □

Combining the criteria of lemma 2 with the one of [5], we obtain a sufficient condition for guaranteeing correctness while moving from SC to TSO to PSO. The condition is expressed on executions, but can easily be mapped to a condition on programs: in the control graph of the program, an *mfence* (resp. *sfence*) must be inserted on all paths leading from a store to a load (resp. a store to a store accessing different variables). This is sufficient, but can insert many unnecessary *mfence*/*sfence* instructions. We now turn to an approach that aims at only inserting the fence instructions that are needed to correct errors that have actually appeared when moving the program from SC to TSO to PSO.

5 An iterative fence insertion algorithm

The basic outline of the algorithm is quite simple: it consists of two steps, and is based on the iterative algorithm of [5]:

1. apply the iterative algorithm of [5] for TSO, starting with a safe program P under SC and returning a TSO-safe program P', by inserting only mfence instructions into the program;
2. apply the iterative algorithm of [5] adapted as described below for PSO, starting with the TSO-safe program P' and returning a PSO-safe program P'', by inserting only sfence instruction into the program.

The algorithm will thus first make the program correct under TSO by iteratively inserting mfence operations. When this is done, the TSO-safe program is analyzed under PSO, and sfence operations are inserted iteratively until the program is correct under PSO. Both parts are guaranteed to terminate, see [5] for the first step and lemma 2 for the second step.

In this second step, the idea is still to look for relaxations (this time we look for *store-store* relaxations) that occur on a path that leads to an error state. To detect *store-store* relaxations, we need to keep track of which operations are compatible with TSO and which are not. This is done by running the state-space exploration with TSO store buffers alongside the PSO store buffers. All operations are also applied to the TSO-buffers, until a *store-store* relaxation is encountered. Once such a relaxation is encountered, we stop updating the TSO-buffer for the process for which the relaxation has occurred since the execution no longer is a TSO-execution, while continuing to update the TSO-buffers for the other processes. Note however that once the TSO-buffer stops being updated for a process, updating can be restarted when all PSO-buffers of that process are completely empty, the TSO-buffer being then reset to empty.

A *store-store* relaxation is detected as follows. The set of enabled transitions of a given global state is computed using the PSO-buffers, which allows the memory order of stores to be changed. When the order of two stores is changed, i.e. a commit of a store is executed while an earlier store accessing another variable is still in the corresponding buffer, the commit of the later store cannot be executed on the TSO-buffer, which indicates that a relaxation has occurred, and the state is marked as a *store-store* relaxation. This relaxation can be disabled by placing an sfence operation before the store operation for which the infringing commit has been executed.

When exploring the state-space under PSO, we know that, if we reach an error state, at least one *store-store* relaxation must have occurred on the path leading to that state. It is then sufficient to disable one of these relaxations to remove that path. When there is a choice of relaxations to disable, we choose the latest on the path leading to the detected error state.

Remark 1. Note that we will not necessarily detect all *store-store* relaxations on a path, as our symbolic buffer content representation makes it impossible to keep the TSO-buffer correctly updated once a relaxation has occurred. New iterations will thus be necessary to find all *store-store* relaxations.

Remark 2. The algorithm we have presented does not guarantee that a program with a minimal number of fences is produced. It could happen that, after the algorithm has iteratively inserted a given number of fences, a fence that was inserted becomes unnecessary due to fences inserted later. One could reiterate on the introduced fences by removing a fence and checking if an error state can be reached. If so, the fence is needed, if not, we can safely remove it. After repeating this procedure until no more fences can be removed we obtain a fence set called "*maximal permissive*"[2], meaning that each fence is needed to ensure the safety property. This does not however imply that the set of inserted fences is globally minimal since the set obtained is dependent on the order in which fences are inserted.

Note however, that no inserted sfence can make an mfence unnecessary. Indeed, sfences will not prevent the *store-load* relaxations that can occur in TSO. The reiteration for removing unnecessary fences should thus be done first after inserting mfences to make the program correct under TSO, and then a second time after the insertion of sfences to adapt the program for PSO.

6 Experimental Results

The fence insertion technique presented in this paper has been implemented within the prototype tool described in [4], extending the tool presented in [5]. The input language for this tool is a simplified and modified version of Promela. It is implemented in Java and uses the BRICS automata-package [14] for handling the automata representing buffer contents.

[2] which was first defined in [13]

This prototype has been tested on examples, most of which are mutual exclusion algorithms (part (a) of Tab. 1). For all those algorithms, we could successfully modify the programs to produce a PSO-safe program, by first iteratively inserting mfence operations in order to make the program TSO-safe, followed by iteratively inserting sfence operations to finally obtain a PSO-safe version of the program. For all those programs, no limitation on the size of buffers were enforced, and most of the programs were analyzed when all processes try to enter into the critical section repeatedly. The only program where only a single entry by each process were considered is Lamport's Bakery, where the use of the counter pushes the repeated entry version beyond the scope of our tool. For those programs, the number of iterations is the sum of inserted mfences and sfences, incremented by 2 (each step (for TSO and then PSO) needs an iteration to build the state-space of the corrected program and check that there are no more errors). The column #St contains the number of states in the state-space of the corrected program (all mfences and sfences inserted). All computed fence sets are maximal permissive, except for Szymanski's algorithm and Lamports fast mutex. For both of these algorithms, one could use Remark 2 to obtain a maximal permissive fence set.

Part (b) of Tab. 1 describes the results for programs that were analyzed and did not need to be corrected to stay correct under TSO or PSO. For those programs, execution times only contains one iteration, which explored the state-space under PSO only, without detecting any error state. All those programs were taken from [15], the Increasing Sequence example being limited to 10 instead of 20.

Table 1. Experimental results for several programs with memory fence insertion

Mutual Exclusion Algorithms			Corrected PSO-safe program				
Program	entry-vers	#Proc	#St	#it	#mfence	#sfence	t
Dekker	repeated	2	381	6	4	0	1.9s
Peterson	repeated	2	219	6	2	2	1.4s
Generalized Peterson	repeated	3	28544	8	3	3	56.7s
Lamport's Bakery	single	2	727	8	4	2	3.2s
Burns	repeated	2	123	4	2	0	1.2s
Szymanski	repeated	2	221	8	6	0	2.2s
Dijkstra	repeated	2	879	4	2	0	3.9s
Lamport's Fast Mutex	repeated	2	5654	10	4	4	11.3s

(a)

Other programs (PSO-safe)			No fences inserted				
Program	limit	#Proc	#St	#it	#mfence	#sfence	t
Alternating bit	-	2	1184	1	0	0	2.3s
Clh queue lock	-	2	3004	1	0	0	2.8s
Increasing Sequence	10	2	59570	1	0	0	140s

(b)

All experimental results were obtained by running our Java-program on a laptop with a 2.7GHz quad-core processor and 8GB RAM, running Ubuntu.

7 Conclusions and comparison with other work

Other work on verification under relaxed memory models includes [16], which proceeds by detecting behaviors that are not allowed by SC but might occur under TSO (or *PSO*). This is done by only exploring SC interleavings of the program, and by using explicit store buffers. The more theoretical work presented in [17] uses results about systems with lossy fifo channels to prove the decidability of reachability under TSO (or PSO) with respect to unbounded store buffers, but the undecidability of repeated reachability. Another approach adopts the axiomatic definition of relaxed memory models and exploit SAT-based bounded model checking [18–20], which of course pushes handling cyclic programs or unbounded buffers beyond their reach. Yet a different approach can be found in [21], which proposes an approach based on SPIN that uses a Promela model with (bounded) explicit queues and an explicit representation of the dependencies on memory accesses that are implied by the relaxed model *RMO* (*Relaxed Memory Order*) [3]. Finally, [22] presents an approach for the verification of programs under relaxed memory models where finite-state programs under SC may turn into infinite-state programs under TSO that proceeds by under-approximation.

With respect to fence insertion algorithms, several other approaches have been proposed. Note that the main originality of our approach is that it is based on a tool that can analyze cyclic programs under TSO/PSO and thus that it can infer fence insertion in this context.

In [23], an over-abstraction technique for potentially infinite store buffers is proposed, combined with the fence insertion algorithm described as "maximal permissive" that was presented in [13]. The abstraction works by representing the buffers as a combination of a finite fifo-buffer that keeps the order of the stores and of an unordered set of stores that is used when the fifo-buffer is full. The fence inference technique works by propagating through the state graph constraints that represent relaxations that could be removed by an mfence or sfence. Once an undesirable state is reached, one can use the associated constraints in order to determine how to make that state unreachable for all incoming paths. However, even if the state-space that is computed is finite in theory, the number of states grows very fast, even for very simple programs, which puts Lamport's fast mutex out of reach of this method, if a first fence is not manually inserted before running the tool. A version of the CLH queue lock could also not be handled, but it is unclear if their version and ours are the same. Also, the increasing sequence example cannot be verified by their approach. For all programs that both our and their approach can handle, and for which no manual fence insertion was done, the computed fences are the same.

Another important piece of work to mention is [15], which exploits the fact that TSO can be simulated by lossy fifo channels. The advantage is that in this setting, state reachability is decidable by a procedure that can be implemented

quite efficiently. This approach, combined with a fence insertion algorithm that computes all minimal fence sets, by restricting the places in the program where fence insertion is allowed, makes it very efficient in the case of TSO. It is worth mentioning that their technique for computing the minimal fence sets is compatible with our approach in the case of TSO, as they iteratively construct those sets by looking for relaxations on a path to an error state. However, in the case of TSO, our approach for inserting mfences iteratively is as optimal as the one in [15], and the number of mfences is consistent with their results. It might be confusing that for Dekker's algorithm, we insert 4 instead of 2 mfences, but this is only caused by a different modeling of the same algorithm.

Finally, the simultaneously appearing [24] presents results based on the idea of "TSO-Robustness", i.e. ensuring by fence insertion that, under TSO, only executions which correspond to SC-executions are allowed. It does not consider PSO.

As conclusion, we have successfully extended our previous work on relaxed memory models from TSO to PSO, obtaining experimental results that compare favorably with other results on this topic. It came as a pleasant surprise while developing these results that the synchronized writing to different buffers that at first seemed necessary and impossible to handle simply, was in fact not needed.

References

1. Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Trans. Computers **28**(9) (1979) 690–691
2. SPARC International, Inc., C.: The SPARC architecture manual: version 8. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1992)
3. SPARC International, Inc., C.: The SPARC architecture manual (version 9). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1994)
4. Linden, A., Wolper, P.: An automata-based symbolic approach for verifying programs on relaxed memory models. In: Proceedings of the 17th international SPIN conference on Model checking software. SPIN'10, Berlin, Heidelberg, Springer-Verlag (2010) 212–226
5. Linden, A., Wolper, P.: A verification-based approach to memory fence insertion in relaxed memory systems. In: Proceedings of the 18th international SPIN conference on Model checking software, Berlin, Heidelberg, Springer-Verlag (2011) 144–160
6. Boigelot, B., Godefroid, P., Willems, B., Wolper, P.: The power of qdds (extended abstract). In: Proceedings of the 4th International Symposium on Static Analysis. SAS '97, London, UK, UK, Springer-Verlag (1997) 172–186
7. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem. Volume 1032 of Lecture Notes in Computer Science. Springer (1996)
8. Loewenstein, P., Chaudhry, S., Cypher, R., Manovit, C.: Multiprocessor memory model verification. Technical report Unpublished presentation at FLOC-AFM 2006, `http://fm.csl.sri.com/AFM06/papers/4-Loewenstein.pdf`.
9. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53** (July 2010) 89–97

10. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-tso. In: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics. TPHOLs '09, Berlin, Heidelberg, Springer-Verlag (2009) 391–407
11. Holzmann, G.: Spin model checker, the: primer and reference manual. First edn. Addison-Wesley Professional (2003)
12. Boigelot, B., Wolper, P.: Symbolic verification with periodic sets. In: Proceedings of the 6th International Conference on Computer Aided Verification. CAV '94, London, UK, UK, Springer-Verlag (1994) 55–67
13. Kuperstein, M., Vechev, M., Yahav, E.: Automatic inference of memory fences. In: Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design. FMCAD '10, Austin, TX, FMCAD Inc (2010) 111–120
14. Møller, A.: dk.brics.automaton – finite-state automata and regular expressions for Java (2010) `http://www.brics.dk/automaton/`.
15. Abdulla, P.A., Atig, M.F., Chen, Y.F., Leonardsson, C., Rezine, A.: Counter-example guided fence insertion under tso. In: Proceedings of the 18th international conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS'12, Berlin, Heidelberg, Springer-Verlag (2012) 204–219
16. Burnim, J., Sen, K., Stergiou, C.: Sound and complete monitoring of sequential consistency for relaxed memory models. In: Proceedings of the 17th international conference on Tools and algorithms for the construction and analysis of systems. TACAS'11/ETAPS'11, Berlin, Heidelberg, Springer-Verlag (2011) 11–25
17. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '10, New York, NY, USA, ACM (2010) 7–18
18. Burckhardt, S., Alur, R., Martin, M.M.K.: Checkfence: checking consistency of concurrent data types on relaxed memory models. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. PLDI '07, New York, NY, USA, ACM (2007) 12–21
19. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: Proceedings of the 20th international conference on Computer Aided Verification. CAV '08, Berlin, Heidelberg, Springer-Verlag (2008) 107–120
20. Burckhardt, S., Alur, R., Martin, M.M.K.: Bounded model checking of concurrent data types on relaxed memory models: a case study. In: Proceedings of the 18th international conference on Computer Aided Verification. CAV'06, Berlin, Heidelberg, Springer-Verlag (2006) 489–502
21. Jonsson, B.: State-space exploration for concurrent algorithms under weak memory orderings: (preliminary version). SIGARCH Comput. Archit. News **36** (June 2009) 65–71
22. Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in tso analysis. In: Proceedings of the 23rd international conference on Computer aided verification. CAV'11, Berlin, Heidelberg, Springer-Verlag (2011) 99–115
23. Kuperstein, M., Vechev, M., Yahav, E.: Partial-coherence abstractions for relaxed memory models. In: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation. PLDI '11, New York, NY, USA, ACM (2011) 187–198
24. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness agains tso. In: Proceedings of the 22nd European conference on Programming Languages and Systems. ESOP'13 (2013) to appear