A Verification-Based Approach to Memory Fence Insertion in Relaxed 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 TSO (Total Store Order) relaxation, which corresponds to the use of store buffers, and its extension x86-TSO, which in addition allows synchronization and lock operations.
The proposed approach uses 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 memory model, but that might be incorrect under x86-TSO. This program is then analyzed for this relaxed memory model and when errors are found (with respect to safety properties), memory fences are inserted in order to avoid these errors. The approach proceeds iteratively and heuristically, inserting memory fences until correctness is obtained, which is guaranteed to happen.
An advantage of our technique is that the underlying symbolic verification tool makes a full exploration 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

Model-checking tools such as SPIN [1] verify concurrent programs under the traditional *Sequential Consistency* (*SC*) memory model [2], in which all accesses to the shared memory are immediately visible globally. However, modern multiprocessor architectures implement relaxed memory models, such as *Total Store Order* (*TSO*) [3, 4], which allow many more possible executions and thus can introduce errors that were not present in SC. Of course, one can force a program executed in the context of TSO to behave exactly as in SC by adding synchronization operations after every memory access. But this totally defeats

* This work is supported by the *Interuniversity Attraction Poles* program *MoVES* of the Belgian Federal Science Policy Office, and by the grant 2.4545.11 of the Belgian Fund for Scientific Research (F.R.S.-FNRS).

the performance advantage that is precisely the motivation for implementing relaxed memory models, rather than SC. Thus, when moving a program to an architecture implementing a relaxed memory model (which includes most current multi-core processors), it is essential to have tools to help the programmer check if correctness is preserved and, if not, to minimally introduce the necessary synchronization operations.

Processor vendors do not publish formal definitions of the memory models their products implement, but rather document the memory model by describing sets of typical behaviors. This is not sufficient for building verification tools, but the problem has been well studied and quite simple models that cover the behaviors that can be seen in actual processors have been defined. These models can be axiomatic, giving constraints that characterize the possible behaviors, or operational, giving a program like description of the relaxed memory system. The model we will use is TSO [3, 4], and its extension *x86-TSO* [5], which covers the possible behaviors of many current processors. In x86-TSO, just as in TSO, writes are buffered, and each processor can read its last buffered values before these become globally visible. In addition, x86-TSO includes a memory barrier instruction (*memory fence*) that can be used for synchronization, as well as a *global lock* allowing atomic operations on the shared memory. The operational model of x86-TSO is quite natural: each process writes to its own buffer from which it can also read, writes being nondeterministically committed to main memory.

Based on these models, several verification approaches and tools for concurrent programs executed under relaxed memory models have been developed. In [6], we proposed a technique that incorporates the TSO store buffers into the model and uses finite automata to represent the possibly infinite set of possible contents of these buffers. This representation coupled with acceleration techniques similar to those proposed in [7], as well as with the *sleep-sets* partial-order reduction techniques [8], allows a full exploration of the state space of programs, including cyclic programs. Other work on this topic includes [9], which proceeds by detecting behaviors that are not allowed by SC but might occur under TSO (or *PSO* (*Partial Store Order*) [4]). This is done by only exploring SC interleavings of the program, and by using explicit store buffers. The more theoretical work presented in [10] uses results about systems with lossy fifo channels to prove the decidability of reachability under TSO with respect to unbounded store buffers, but the undecidability of repeated reachability. Other approaches to verification, with respect to relaxed memory models, adopt the axiomatic definition of these models and exploit SAT-based bounded model checking [11, 12], which of course pushes handling cyclic programs or unbounded buffers beyond their reach. Finally, [13] 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*) [4].

This paper focuses on porting a program, verified under SC to x86-TSO, while preserving its safety properties. The approach is based on the verifica-

tion tool presented in [6], which makes it applicable to cyclic programs. A first contribution is to show that the performance of this tool can be improved by a more complete use of partial-order techniques. The improvement is such that the number of states explored while verifying under the relaxed memory model is often not significantly larger than the number of states explored in a verification under SC.

The second contribution of the paper is the method developed for safely porting programs from SC to x86-TSO. It starts by attempting to verify the program under x86-TSO and, if an undesirable state is found to be reachable, memory fences are inserted. The insertion policy is based on the observation that if no process can execute a *load* after a *store* without going through an *mfence* (memory fence), then every safety (unreachability of undesirable states) property satisfied under SC is also satisfied under x86-TSO. Exploiting this, we develop a heuristic iterative approach, which is guaranteed to converge.

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 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:

- $store(p_i, m_j, v)$, the meaning of which is that process p_i stores the value $v \in \mathcal{D}$ to the memory location m_j,
- $load(p_i, m_j, v)$, the meaning of which is that process p_i loads the value stored in memory location m_j 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 can lead to executions that are not possible on a sequential consistency memory system. For example, in the program given in Table 1, both processes can run through and finish their execution if run on a TSO memory system, but this

cannot happen on an SC memory system, where at least one process would find the value 1 in its last load operation, which would thus not go through.

Table 1. Intra-processor forwarding example from [14]

initially: `x = y = 0;`	
Process 1	Process 2
$store(p_1, \mathtt{x}, 1)$ (s_1)	$store(p_2, \mathtt{y}, 1)$ (s_2)
$load(p_1, \mathtt{x}, 1)$ (l_1)	$load(p_2, \mathtt{y}, 1)$ (l_3)
$load(p_1, \mathtt{y}, 0)$ (l_2)	$load(p_2, \mathtt{x}, 0)$ (l_4)

The x86-TSO memory model [15, 5] is an extension of TSO in which operations such as *atomic writes* and *flushing* (emptying) *the buffer content* to the shared memory, have been added. In spite of its name, x86-TSO is not an exact model of a given architecture, but is an abstract programmer's model that covers the documented behaviors of a wide range of processors. One can thus safely assume that a program verified under x86-TSO will run correctly on the corresponding processors.

The formal definitions of the memory models use the concepts of *program order* and *memory order* [3, 16]. 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 op denote any load or store operation, l any load operation, s any store operation, l_a a load operation on location a, and s_a a store operation on location a. Furthermore, let $val(l)$ be the value returned by the load operation l and let $val(s)$ be the value stored by the store operation s.

An SC execution is one for which there exists a memory order satisfying the following constraint for each process p_i:

1. $\forall op_i, op_j : op_i <_p op_j \Rightarrow op_i <_m op_j$

This means that the memory order is an interleaving of the program orders. The result of a load operation is then simply the value of the most recent (in memory order) store to the same location.

On the other hand, a TSO execution is then one for which there exists a memory order satisfying the following constraints:

1. $\forall l_a, l_b : l_a <_p l_b \Rightarrow l_a <_m l_b$
2. $\forall l, s : l <_p s \Rightarrow l <_m s$
3. $\forall s_a, s_b : s_a <_p s_b \Rightarrow s_a <_m s_b$
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 load that is executed within the same process later than the store, this is known as the $store \rightarrow load$ order relaxation. 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.

For the example of Table 1, a possible memory order is given in Table 2. The first process starts its execution by its store operation, which is not directly added to the memory order. The following load operations are directly added to the memory order, and the first process finishes its execution without being blocked. Then, the second process starts its execution with its store operation, which is not directly added to the memory order. The following load operations are then executed and added to the memory order. Finally, the stores of both processes are eventually transferred to main memory and introduced in the memory order.

Table 2. Possible operation sequence and memory order

Operation sequence	Associated memory orderings
$store(p_1, \mathtt{x}, 1) \quad (s_1)$	-
$load(p_1, \mathtt{x}, 1) \quad (l_1)$	l_1
$load(p_1, \mathtt{y}, 0) \quad (l_2)$	$l_1 <_m l_2$
$store(p_2, \mathtt{y}, 1) \quad (s_2)$	$l_1 <_m l_2$
$load(p_2, \mathtt{y}, 1) \quad (l_3)$	$l_1 <_m l_2 <_m l_3$
$load(p_2, \mathtt{x}, 0) \quad (l_4)$	$l_1 <_m l_2 <_m l_3 <_m l_4$
-	$l_1 <_m l_2 <_m l_3 <_m l_4 <_m s_1$
-	$l_1 <_m l_2 <_m l_3 <_m l_4 <_m s_1 <_m s_2$

The characterization of TSO we have just given is useful in dealing with TSO axiomatically, but not adapted for applying state-space exploration verification techniques. Fortunately, there exists a natural equivalent operational description of TSO, as well as of x86-TSO. In this description (see Fig. 1), stores from each process are buffered and eventually transferred to shared memory in an interleaved way. A store only takes effect globally when it is transferred (or committed) to shared memory, which is also when it is introduced into the memory order. Committing to shared memory is an internal operation, which is assumed to be executed nondeterministically. When a process executes a load, it will read the most recent value in its own store buffer or, if there is no such buffered store, the value found in shared memory.

In x86-TSO, a global lock is added to enable the processes to gain exclusive access to the shared memory. While this global lock is held by a given process, no other process can obtain the lock or execute a load operation, and the only commit operations that can be executed are those of the process holding the lock.

An *unlock* operation is only possible if the process executing it holds the lock and if the store buffer of that process is empty. Additionally, a new operation called *mfence* is added to the set of operations. An mfence operation simply blocks the process executing it until its store buffer is empty.

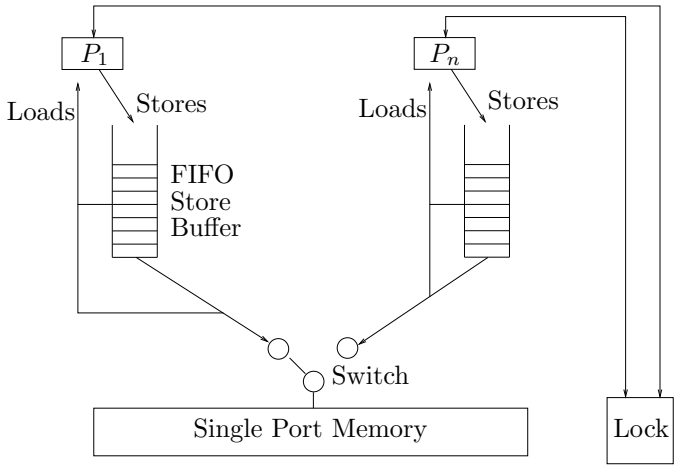

Fig. 1. Operational definition of x86-TSO of [5]

The formal operational model of x86-TSO is obtained by extending the program as follows. A set

$$\mathcal{B} = \{b_{p_1}, \ldots, b_{p_n}\}$$

of store buffers are introduced, one for each process[1]. One also adds a global lock L whose value can be a process p, or undefined ($\perp$). A global state is thus the composition of the content of the memory, the value of the global lock, and, for each process p, a control location and the content of its store buffer $[b_p]$, which is a word in $(\mathcal{M}, \mathcal{D})^*$.

The precise semantics of the operations can then be described as follows.

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

$$[b_p] \leftarrow [b_p](m, v).$$

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

If ($[L] \neq \perp$ and $[L] \neq p$), then $load(p, m, v)$ cannot be executed;

[1] Note that we introduce a buffer 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.

otherwise, let $[b_p] = (m_1, v_1)(m_2, v_2)\dots(m_f, v_f)$ and let $i = \max\{j \in \{1\dots f\} \mid m_j = m\}$. If i exists, then the result of the load is the test $d_i = d$. If not, it is the result of the test $[m] = v$, where $[m]$ denotes the content of the memory location m.

mfence operation : *mfence*(p):

If ($[b_p] = \varepsilon$) then *mfence*(p) is enabled;
otherwise *mfence*(p) cannot be executed.

lock operation : *lock*(p):

If ($[L] = \perp$ or $[L] = p$) then *lock*(p) is enabled;
otherwise, *lock*(p) cannot be executed.

unlock operation : *unlock*(p):

If ($[L] = p \wedge [b_p] = \varepsilon$) then $[L] \leftarrow \perp$ (the lock is released);
otherwise *unlock*(p) cannot be executed.

commit operation : *commit*(p):

If ($[L] \neq \perp$ and $[L] \neq p$), then *commit*(p) cannot be executed;
otherwise, let $[b_p] = (m_1, v_1)(m_2, v_2)\dots(m_f, v_f)$. Then, if $[b_p] \neq \varepsilon$, the result of the commit operation is

$$[b_p] \leftarrow (m_2, v_2)\dots(m_f, v_f)$$

and

$$[m_1] \leftarrow v_1, \text{ or}$$

if $[b_p] = \varepsilon$, the commit operation has no effect.

Note that *commit(p)* 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)* or *unlock(p)* operation is blocked because the process buffer is nonempty, the implicit execution of *commit(p)* operations makes it possible to empty the buffer and enable this operation.

3 Representing Sets of Buffer Contents and State Space Exploration

Verifying a program under the x86-TSO memory model can be done with a tool such as SPIN. 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 [6] as follows. To start with, rather than limiting buffers to a fixed size, finite automata are used to represent possibly infinite sets of buffer contents. This allows unbounded buffer contents to be taken into account and, with the help of acceleration techniques similar to those of [17] and [7], to explore the full state space of programs, even if they include memory accesses (especially memory writes) in cycles that can be indefinitely repeated.

This approach, with the help of *partial-order* techniques, was also quite helpful in coping with the problem of the size of the state space. Indeed, the automata representing buffer contents stay reasonably small, since their size is only of the order of the one of the program for the process writing to the buffer. Furthermore, buffering memory write operations introduces a lot of independence between the actions of the various processes, which makes the use of partial-order reduction techniques especially effective. This was exploited in [6] by using *sleep-sets*, as well as other optimization such as only executing commit operations just before operations with which they are dependent, namely load, mfence and unlock operations.

The version of the tool used for this paper goes further and fully implements the *persistent-sets* reduction of [8, 18]. Persistent sets are formally defined as follows.

Definition 1. *A set* T *of transitions enabled in a state* s *is* persistent in s *iff, for all nonempty sequences of transitions*

$$s = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \dots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$$

from s *in the transition system and including only transitions* $t_i \notin T, 1 \leq i \leq n, t_n$ *is independent in* s_n *with all transitions in* T.

Our implementation uses a simple greedy algorithm for computing persistent sets and handles the *ignoring problem*[2] described in [19, 8], by using a *proviso condition* as suggested in these references. The proviso condition we use imposes that, if none of the transitions to be explored from a state leads to a state that is not on the current search stack, then all transitions from that state must be explored. However, this only guarantees the reachability of local states, some global states potentially being left unexplored. Thus, to force the detection of global error states, we consider as dependent with transitions of other processes all transitions leaving a local state that is part of an error state.

The procedure to compute a persistent set satisfying the proviso condition is the following. One searches for a process whose enabled operations are only store or local operations and satisfy the proviso condition, i.e. contain at least one operation leading to a state that is not on the search stack. If furthermore the local state of this process is not part of a global error state, the persistent set is taken to be the set of enabled transitions of this process. Indeed, this set

[2] A partial-order search might "ignore" a process and thus leave it totally inactive at some point.

will be independent with all operations of the other processes, as well as with respect to all possible commit operations. If such a process cannot be found, the persistent set is taken to be the whole set of enabled transitions (including commits).

4 From SC to TSO

We turn to the problem of preserving the correctness of a program when it is moved from an SC to an x86-TSO 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 x86-TS0 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, this is more than is necessary to guarantee that the executions that can be seen under x86-TSO are also possible under SC. Recall that the difference between the axiomatic definition of SC and of TSO is the absence of the following store-load constraint in TSO:

$$\forall l, s : s <_p l \Rightarrow s <_m l \tag{1}$$

Thus stores can be postponed in memory order, which leads to executions that are not possible in SC. To avoid this it is sufficient to make sure that no process can execute a load after a store without going through an mfence. Indeed, even though successive stores might be buffered, they will be committed to main memory before any later load and hence the constraint (1) will be satisfied by the memory order, just as in SC. The memory order then becomes an interleaving of the program orders and the execution semantics thus match SC. We formalize this in the following lemma.

Lemma 1. *Given an x86-TSO execution, if in the program order of each process, an* mfence *is executed between each* load *and any preceding* store*, the memory order satisfies all the SC constraints.*

Proof. The semantics of mfence operations can be formalized by introducing these operations in the memory order with the following constraints, s, l and m representing store, load, and mfence operations respectively:

1. $\forall s, m : s <_p m \Rightarrow s <_m m$
2. $\forall s, m : m <_p s \Rightarrow m <_m s$
3. $\forall l, m : l <_p m \Rightarrow l <_m m$
4. $\forall l, m : m <_p l \Rightarrow m <_m l$

In the conditions of the lemma, we have that if $s <_p l$, there is an mfence m such that $s <_p m$ and $m <_p l$. And thus we have that $s <_m l$, using the semantics of memory fences.

The memory order thus satisfied all constraints of an SC order.

Remark 1. Using Lemma 1, one can deduce that for any algorithm that writes to memory exclusively with processor instructions that include an implicit memory barrier (such as *CAS* (compare-and-swap) or *TAS* (test-and-set)), moving this algorithm from an SC to a TSO architecture will preserve its correctness.

The Michael-Scott non-blocking queue [20] is such an algorithm, in which all store operation to shared memory are implemented by CAS operations, and thus will run correctly under x86-TSO. This is consistent with the results in [21].

We now have a sufficient condition for guaranteeing correctness while moving from SC to x86-TSO. 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 must be inserted on all paths leading from a store to a load. This is sufficient, but can insert many unnecessary mfence instructions. We now turn to an approach that aims at only inserting the mfence instructions that are needed to correct errors that have actually appeared when moving the program to x86-TSO.

5 An iterative mfence insertion algorithm

Our method is quite simple:

- The explicit state verification algorithm, presented in [6] and extended with the *persistent-set* partial-order reduction, is run until a state violating a correctness criterion is found;
- A search for a place in which to insert an mfence in order to make the undesirable state unreachable is performed and the mfence instruction is added to the program;
- The procedure is repeated until no further undesirable state can be reached.

The central algorithm is Algorithm 2. It is the algorithm used in [6] with a call to the function *insertMfence()* being executed when an incorrect state is reached. When this happens the function *DFS()* returns *false*, which is passed back up the recursive call chain. Otherwise, *DFS()* returns *true*.

The main program is presented in Algorithm 1. It simply initializes the search and repeats it until *DFS()* returns *true*.

We now describe the method used in the *insertMfence()* procedure. Since we started with an algorithm that is correct under SC, an undesirable state can only be reached because of the relaxed memory model. Comparing x86-TSO and SC, and using the same line of reasoning as that leading to Lemma 1, this can only happen if a load is performed when the corresponding buffer is nonempty.

Thus, the procedure *insertMfence()* starts from the detected error state and searches backwards through the current search path for such a situation. We

Algorithm 1 Iterative initialization and call of depth-first search with error handling

```
1: repeat
2:    init(Stack)
3:    init(H) /* Table of visited states */
4:    s_0 = initial state
5:    s_0.Sleep = ∅
6:    delay(s_0) = ∅
7:    push s_0 onto Stack
8: until (DFS())
```

could directly insert an mfence just before the offending load in the code of the process executing it, but this would be suboptimal if the previous instruction was a load and not a store given that only store to load transitions are problematic. The backwards search is thus continued until a store executed by the same process as the offending load is found. When this operation is detected, we insert an mfence operation right after it in the control graph of the process.

Note that we just insert one mfence at each run of the verification procedure. This means that the procedure will usually be run repeatedly, but since the number of possible mfence insertions is bounded by the program size, the iterative process will always terminate. Moreover, as we only insert necessary mfence operations, the number of fence instructions inserted is in this sense optimal, but the optimal is local: there is no proof that we always reach a globally minimal number of mfence insertions.

6 Experimental Results

The memory fence insertion technique presented in this paper has been implemented within the prototype tool described in [6]. The input language for this tool is a simplified version of Promela. It is implemented in Java and uses the BRICS automata-package [22] for handling the automata representing buffer contents.

This prototype has been tested on examples, most of which are mutual exclusion algorithms: Dekker's and Peterson's algorithm for mutual exclusion, and Lamport's Bakery algorithm. We also considered all the litmus tests proposed in [5] together with the definition of x86-TSO. These litmus tests are sample programs provided by processor vendors for illustrating possible behaviors of the memory system. Our tool was used in several ways on these litmus tests. First, it was checked that the possibility/impossibility scenarios given for the litmus tests fell respectively within/outside the behaviors considered possible by our tool. The second was to consider non SC behaviors allowed by the litmus tests to be errors and to use the approach of this paper to insert memory fences in order to eliminate these behaviors. All this was performed successfully.

For mutual exclusion, both a single entry version and a repeated entry version were considered for Dekker's and Peterson's algorithms. In the single entry

Algorithm 2 Recursive DFS() procedure with error detection and mfence insertion

```
 1: s = top(Stack)
 2:
 3: /* if s is an error state: */
 4: /*   search for the last relaxation in the current search path */
 5: /*   and insert a mfence to avoid this relaxation, and return false */
 6: if (s is an error state) then
 7:    insertMfence()
 8:    return false
 9: end if
10:
11: /* Go through stack from top to bottom, looking for cycles */
12: for all ss in (Stack \ top(Stack)) do
13:    if (cycleCondition(ss,s)) then
14:       s = cycle(ss,s)
15:       break
16:    end if
17: end for
18:
19: if (∃sI ∈ H | s ⊆ sI) then
20:    iSleep = ⋂_{∀sI∈H|s⊆sI} H(sI).Sleep
21:    T = {t | t ∈ iSleep ∩ t ∉ s.Sleep}
22:    s.Sleep = s.Sleep ∩ iSleep
23:    if (s ∈ H) then
24:       H(s).Sleep = s.Sleep
25:    else
26:       enter s in H
27:    end if
28: else
29:    enter s in H
30:    T = Persistent_Set_satisfying_Proviso(s)\s.Sleep
31: end if
32:
33: for all t ∈ T do
34:    ssucc = succ(s,t)
35:    ssucc.Sleep = {tt | tt ∈ s.Sleep ∧ (t, tt) independant in s}
36:    push ssucc onto Stack
37:
38:    /* if an error is encountered, return false */
39:    if (!DFS()) then
40:       return false
41:    end if
42:
43:    s.Sleep = s.Sleep ∪ {t}
44: end for
45: pop(Stack)
46: return true
```

version, each process only attempts to enter the critical section once, whereas in the repeated entry version, each process repeatedly attempts to enter the critical section. A single entry version of the generalized Peterson's algorithm with 3 processes was also analyzed, as well as Lamport's Bakery with 2 processes. Also notice that the loop executed in that protocol was unrolled, which explains why 6 locations for memory fence insertion are found, rather than the 3 required. Given this, the number of mfence operations operations inserted by our algorithm is, for all the examples we have handled, optimal.

For Lamport's Bakery algorithm, the counter used pushes the repeated entry version beyond the scope of our tool.

The results with and without error correction are given in Table 3. Column 2 defines the entry version (single or repeated), Column 3 gives the number of processes. Columns 4 and 5 give information (number of states and time respectively) about the state-space exploration when no fences are inserted. Columns 6 to 9 give information about the exploration with error correction. Column 7 gives the number of iterations needed to insert enough memory fences to correct the program (and do a last check of its correctness) and Column 6 gives the number of states in the final program. The 8th Column gives the number of memory fences inserted, and the last Column gives the total amount of time needed to insert iteratively the fences and to finally verify that the safety property holds again. Very interestingly, even though it involves several iterations, this time is, in almost every case, lower than the time needed to explore the full state space of the uncorrected version.

For Dekker's algorithm, it is important to work with the repeated entry version, since with the single entry version, only 2 memory fences need to be inserted, whereas 4 are essential for the repeated entry version. Note that in [21], only 2 memory fences for Dekker under TSO are detected. This appears to be the linked to the fact that they are using the version of Dekker's algorithm given in appendix J of [4], where it is classified as not deadlock-free. The version we consider is deadlock-free and guarantees freedom of non-progress cycles.

Table 3. Experimental results for Dekker's and Peterson's Algorithm for mutual exclusion and for Lamport's Bakery with and without memory fence insertion

			without err. corr.		with err. corr.			
Program	entry-vers	#Proc	#St	t	#St	#it	#f	t
Dekker	single	2	118	0.84s	92	3	2	0.80s
Dekker	repeated	2	5468	12.70s	213	5	4	0.41s
Peterson	single	2	108	0.09s	52	3	2	0.03s
Peterson	repeated	2	400	0.58s	54	3	2	0.05s
Gen. Peterson	single	3	15476	44.42s	1164	7	6	1.55s
Lamport's Bakery	single	2	775	0.58s	340	5	4	0.15s

All experimental results were obtained by running our Java-program on a 2.0GHz Intel Core Duo laptop running Ubuntu Linux.

Interestingly, compared to results that can be obtained for the SC memory model, our results show that the exploration of a state space considering the TSO memory model can be performed with a limited increase in the number of states to be explored. Table 4 compares the size of the state-space computed by SPIN (with partial-order reduction) for the SC memory model to the size of the state space computed by our prototype for the x86-TSO memory model, both when the full state-space is explored and when error correction is applied. It might seem surprising that in the latter case, the number of stored states is sometimes smaller than when doing verification under SC. However, this is due to the combination of the partial-order techniques and of the additional independence between the actions of the various processes that comes from delaying stores until they are needed for a load, an mfence or an unlock operation.

Table 4. Comparison of state spaces computed by SPIN for SC and by our implementation for TSO.

			SPIN-SC		Our Prototype-TSO without err. corr.	
Program	entry-vers	#Proc	#St stored	#St visited	#St stored	#St visited
Dekker	single	2	105	165	118	160
Dekker	repeated	2	179	214	5468	11322
Peterson	single	2	22	47	108	134
Peterson	repeated	2	24	49	400	640
Gen. Peterson	single	3	1901	4315	15476	46302
Lamport's Bakery	single	2	238	414	775	1186

			SPIN-SC		Our Prototype-TSO with err. corrected	
Program	entry-vers	#Proc	#St stored	#St visited	#St stored	#St visited
Dekker	single	2	105	165	92	117
Dekker	repeated	2	179	214	213	365
Peterson	single	2	22	47	52	68
Peterson	repeated	2	24	49	54	89
Gen. Peterson	single	3	1901	4315	1164	2697
Lamport's Bakery	single	2	238	414	340	407

Finally, it is worth noting that only one mfence is inserted at each iteration, whereas more could be inserted by matching similar code in the various processes. This could lead to a further reduction of the number of required iterations.

7 Conclusions and comparison with other work

Besides our work, several fence insertion algorithms have been proposed. The main originality of our approach is that it is based on a tool that can analyze cyclic programs under x86-TSO and thus that it can infer fence insertion in this context.

In [21] a fence insertion algorithm described as "maximal permissive", i.e. producing an algorithm with the least possible set of restrictions on its behaviors, is proposed. The approach is based on bounded model-checking. It works by propagating through the state graph constraints that represent relaxations that could be removed by an mfence. Once an undesirable state is reached, one can use the associated constraints in order to determine how to make that state unreachable. This approach cannot be applied to cyclic programs and it is not compatible with partial-order reductions, which does not make it possible to transfer it to our context.

Another automatic fence insertion algorithm, using bounded model-checking is given in [23]. This approach is targeted to programs written in *C#*, whose memory model is more relaxed than SC, and hence can lead to surprises when programs are ported to a non SC environment. They use the *maxflow-mincut* algorithm [24] to decide where to insert memory fences in oder to ensure that error states are not reached. No claim is made about the minimality of the set of inserted fences. Along related lines, [25, 26] considers the *Java* language and aims at preserving its semantics when the program is run under a memory model that is more relaxed then the one specified in the *Java* model.

A less automatic approach was presented in [11], in which the tool could find errors, print the corresponding traces, but leave it to the programmer to decide where to insert memory fences. This approach can only handle finite exploration graphs.

Less directly related work on guaranteeing correct execution under TSO includes [27], which shows that under special conditions, such as the *triangular race freedom* introduced in [28], all behaviors possible in TSO are also possible in SC. However, we are not aware of any automated tool to detect triangular races and achieve freedom of these races. Furthermore, even if a program includes a triangular race this does not imply that it will have incorrect behaviors, with respect to a safety property, under a relaxed memory model.

As conclusions, we first claim that we have shown that with the right combination of techniques (automata for representing buffer contents and partial-order reductions), using explicit state enumeration to verify programs under relaxed memory models can be done with limited penalty compared to verification under SC. Retrospectively, this is not really surprising since using store buffers introduces a lot of independence, which is tamed by the partial-order methods. Our second claim is that we have shown that this can be effectively exploited in order to find which memory synchronization operations need to be introduced to guarantee that correctness is preserved when moving a program from SC to x86-TSO.

References

1. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (September 2003)
2. Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Trans. Computers **28**(9) (1979) 690–691
3. SPARC International, Inc., C.: The SPARC architecture manual: version 8. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1992)
4. SPARC International, Inc., C.: The SPARC architecture manual (version 9). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1994)
5. 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
6. 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
7. Boigelot, B., Godefroid, P., Willems, B., Wolper, P.: The Power of QDDs (Extended Abstract). In Hentenryck, P.V., ed.: Static Analysis, 4th International Symposium, SAS '97, Paris, France, September 8-10, 1997, Proceedings. Volume 1302 of Lecture Notes in Computer Science., Springer (1997) 172–186
8. 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)
9. Burnim, J., Sen, K., Stergiou, C.: Sound and complete monitoring of sequential consistency in relaxed memory models. Technical Report UCB/EECS-2010-31, EECS Department, University of California, Berkeley (Mar 2010)
10. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In Hermenegildo, M.V., Palsberg, J., eds.: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010, ACM (2010) 7–18
11. Burckhardt, S., Alur, R., Martin, M.M.K.: Checkfence: checking consistency of concurrent data types on relaxed memory models. In Ferrante, J., McKinley, K.S., eds.: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007, ACM (2007) 12–21
12. Burckhardt, S., Musuvathi, M.: Effective Program Verification for Relaxed Memory Models. In Gupta, A., Malik, S., eds.: Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings. Volume 5123 of Lecture Notes in Computer Science., Springer (2008) 107–120
13. Jonsson, B.: State-space exploration for concurrent algorithms under weak memory orderings: (preliminary version). SIGARCH Comput. Archit. News **36** (June 2009) 65–71
14. Intel Corporation: Intel®64 and IA-32 Architectures Software Developer's Manual. Specification (2007)
15. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In Berghofer, S., Nipkow, T., Urban, C., Wenzel, M., eds.: Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings. Volume 5674 of Lecture Notes in Computer Science., Springer (2009) 391–407

16. Loewenstein, P., Chaudhry, S., Cypher, R., Manovit, C.: Multiprocessor memory model verification. (2008)
17. Boigelot, B., Wolper, P.: Symbolic Verification with Periodic Sets. In Dill, D.L., ed.: Computer Aided Verification, 6th International Conference, CAV '94, Stanford, California, USA, June 21-23, 1994, Proceedings. Volume 818 of Lecture Notes in Computer Science., Springer (1994) 55–67
18. Godefroid, P., Pirottin, D.: Refining dependencies improves partial-order verification methods (extended abstract). In: Proceedings of the 5th International Conference on Computer Aided Verification. CAV '93, London, UK, Springer-Verlag (1993) 438–449
19. Valmari, A.: Stubborn sets for reduced state space generation. In: Proceedings of the 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets 1990, London, UK, Springer-Verlag (1991) 491–515
20. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing. PODC '96, New York, NY, USA, ACM (1996) 267–275
21. Kuperstein, M., Vechev, M., Yahav, E.: Automatic inference of memory fences. In: Formal Methods in Computer Aided Design. (2010)
22. Møller, A.: brics/automaton DFA/NFA Java implementation.
23. Huynh, T.Q., Roychoudhury, A.: A Memory Model Sensitive Checker for C#. In Misra, J., Nipkow, T., Sekerinski, E., eds.: FM. Volume 4085 of Lecture Notes in Computer Science., Springer (2006) 476–491
24. Ford, L.R., Fulkerson, D.R.: Maxumum flow through a nework. In: Canad. J. Math. Volume 8. (1956) 399–404
25. Sura, Z., Wong, C.L., Fang, X., Lee, J., Midkiff, S.P., Padua, D.A.: Automatic implementation of programming language consistency models. In Pugh, W., Tseng, C.W., eds.: LCPC. Volume 2481 of Lecture Notes in Computer Science., Springer (2002) 172–187
26. Fang, X., Lee, J., Midkiff, S.P.: Automatic fence insertion for shared memory multiprocessing. In: Proceedings of the 17th annual international conference on Supercomputing. ICS '03, New York, NY, USA, ACM (2003) 285–294
27. Attiya, H., Guerraoui, R., Hendler, D., Kuznetsov, P., Michael, M.M., Vechev, M.: Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '11, New York, NY, USA, ACM (2011) 487–498
28. Owens, S.: Reasoning about the implementation of concurrency abstractions on x86-tso. In: Proceedings of the 24th European conference on Object-oriented programming. ECOOP'10, Berlin, Heidelberg, Springer-Verlag (2010) 478–503