An Automata-Based Symbolic Approach for Verifying Programs on Relaxed Memory Models

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 programs for the relaxed memory models implemented in modern processors. Specifically, it considers the TSO (Total Store Order) relaxation, which corresponds to the use of store buffers. The proposed approach proceeds by using finite automata to symbolically represent the possible contents of the store buffers. Store, load and commit operations then correspond to operations on these finite automata.
The advantage of this approach is that it operates on (potentially infinite) sets of buffer contents, rather than on individual buffer configurations. This provides a way to tame the explosion of the number of possible buffer configurations, while preserving the full generality of the analysis. It is thus possible to check even designs that exploit the relaxed memory model in unusual ways. An experimental implementation has been used to validate the feasibility of the approach.

1 Introduction

Modern multiprocessor systems do not implement the traditional *Sequential Consistency* [1] (SC) model of memory access. This fact is usually referred to by describing these processors as implementing *relaxed memory models* that permit executions not allowed in SC. Thus verification tools such as SPIN that are based on the SC model do not reliably verify programs to be run on widely used current processors. It is quite disturbing to observe that even simple mutual exclusion algorithms such as Peterson's do not run correctly on a standard modern multicore computer. This situation is nevertheless mostly hidden from the programmer since process synchronization is done through system provided functions, which are correctly implemented, forcing memory synchronization if needed. This is a safe approach, but leads to a suboptimal use of multicores. Having tools for analyzing programs with respect to the implemented relaxed memory models would be of great help in designing code that does not unduly force synchronization. It would also be most useful for checking that code designed for the SC memory model can be safely ported to processors implementing relaxed memory models or, if needed for minimally correcting such code.

The exact memory model that is implemented varies and deciphering processor documentation on this topic is, to put it mildly, quite challenging. However, the topic is being more and more studied and clear models of memory access models have been proposed. These models can be either axiomatic, giving constraints on possible memory accesses, or operational, giving a program-like description of the shared memory model. Of these models, one of the most studied is the *Total Store Order* (TSO) model. It has a simple axiomatics characterization and a clear equivalent operational description in terms of store buffers. In TSO, processor writes are buffered and each processor reads the last value written to its buffer, while others only see the values committed to main memory. This model was the one implemented in SPARC processors [2] and [3] and closely corresponds to the one implemented in X86 processors [4]. Furthermore, store buffers are an essential ingredient of even more relaxed memory models [5] and thus being able to analyze TSO is an important stepping stone in developing verification tools for relaxed memory models. This paper will thus exclusively focus on TSO.

Since TSO can be modeled by a memory accessed through buffers, an obvious approach to verifying programs under this memory model is to explicitly include the store buffers in the program being analyzed. This has of course already been tried, but requires overcoming two problems. The first is that there is no natural bound on the size of the buffers, the second is the explosion in the number of states due to the introduction of store buffers. For the first problem, one can arbitrarily bound the size of the buffers, which, at best, leads to verification that is unsatisfactorily hardware dependent. For the second problem, various techniques such as SAT based bounded model-checking have been tried with some success [6], but at the cost of limits on what can be verified.

In this paper, we develop an approach inspired by the techniques developed in [7] for verifying systems with unbounded buffers. The main idea is that, since a buffer content can be viewed as a word, sets of buffer contents are languages that can be represented by finite automata. This allows infinite sets of contents to be represented and manipulated by operations on automata. Of course, in a step by step exploration of the state space, infinite sets of buffer contents will never be generated. Acceleration techniques are thus required and these take the form of algorithms for directly computing the possible contents of buffers after repeating a program cycle an arbitrary number of times.

Compared to the general problem of analyzing programs using unbounded buffers, the specific case of TSO buffers offers both simplifications and added difficulties. The main simplification is that each process only writes to a single buffer, which makes a separate representation of each buffer the natural choice. Among the difficulties are the operations on the store buffers, which are not quite like those on communication buffers. Indeed, if a store is very much like a buffer write and a commit to memory is similar to a buffer read, a load operation is special. Indeed, it should retrieve the most recently written value and, when there is a choice of such values, a repeated read should yield an identical result. One of our contributions is thus to define these operations precisely when applied to

sets of store buffer contents and to show how they can be implemented. Another is adapting the cycle iteration acceleration technique to the specific context of store buffers.

To validate our approach we have built an experimental implementation to test the feasibility of the proposed method. Our implementation uses the BRICS automata manipulation package [8] and has allowed us to fully verify (or find errors) in simple synchronization protocols. Since each process writes to its own buffer, the cycle iteration acceleration needs to favor progress by a single process. Partial-order verification techniques [9], and in particular "sleep sets", have been helpful with respect to this. Indeed, it turned out that using sleep sets yielded a significant performance improvement by avoiding the repeated detection of the same cycle from different global control points.

The verification problem we consider has already been addressed in several papers going back at least a decade. In [10] the problem is clearly defined and it is shown that behaviors possible under TSO but not SC can be detected by an explicit state model checker. Later work, [6], uses SAT-based bounded model checking with success for detecting errors with respect to relaxed memory executions. A more recent paper [11] aims at distinguishing programs that can safely be analyzed under SC, even if run in a relaxed memory model environment. Finally, [12] proves decidability and undecidability results for relaxed memory models considering store buffers to be infinite. In this it is very close to our work, but its goal is purely theoretical and it proceeds by reducing the problem to lossy buffer communication. This is very elegant for obtaining decidability results, but of uncertain value for doing actual verification. Indeed, the reduction to lossy buffers implies an elaborate coding of buffer contents. In contrast, our approach works with a direct representation of the store buffer contents and is oriented towards doing actual verification. To our knowledge, it is the first verification technique for relaxed memory models allowing the full generality coming from unbounded store buffer contents.

2 Concurrent Programs and Memory Models

We consider a very simple model of concurrent programs in which a fixed set of finite-state processes interact through a shared memory. A concurrent program is thus defined by a finite set $\mathcal{P} = \{p_1, \ldots, p_n\}$ of processes and a finite set $\mathcal{M} = \{m_1, \ldots, m_k\}$ of memory locations. The memory locations can hold values from a data domain $\mathcal{D}$. The initial content of the memory is given by a function $\mathcal{I} : \mathcal{M} \to \mathcal{D}$.

Each process p_i is defined by a finite set $\mathcal{L}(p_i)$ of control locations, an initial location $\ell_0(p_i) \in \mathcal{L}(p_i)$, and transitions between control locations labeled by operations from a set $\mathcal{O}$. A transition of a process p_i is thus an element of $\mathcal{L}(p_i) \times \mathcal{O} \times \mathcal{L}(p_i)$, usually written as $\ell \xrightarrow{op} \ell'$. The set of operations contains the following memory operations:

- $store(p_i, m_j, d)$, i.e. process p_i stores value $d \in \mathcal{D}$ to memory location m_j (note that since transitions are process specific, mentioning the process in the operation is redundant, but will turn out to be convenient),
- $load(p_i, m_j, d)$, i.e. process p_i loads the value stored in m_j and checks that its value is d. If the stored value is different from d, the transition is not possible.

The SC semantics of such a concurrent program is the usual interleaving semantics in which the possible behaviors are those that are interleavings of the executions of the various processes and in which stores are immediately visible to all processes.

In TSO, each process sees the result of its loads and stores exactly in the order it has performed them, but other processes can see an older value than the one seen by the process having performed a store. This leads to executions that are not possible in SC. For instance, in the program given in Table 1, both processes could terminate their executions, whereas under SC semantics, either p_1 or p_2 will find the value 1 when performing the last load operation. TSO is thus also referred to as the $store \rightarrow load$ order relaxation.

initially: `x = y = 0;`	
Processor 1	Processor 2
$store(p_1, x, 1)$	$store(p_2, y, 1)$
$load(p_1, x, 1)$	$load(p_2, y, 1)$
$load(p_1, y, 0)$	$load(p_2, x, 0)$

Table 1. Intra-processor forwarding, given in [13]

To define TSO formally, one uses the concepts of *program order* and *memory order* [2, 14]. 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 denote 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 $val(l)$ or $val(s)$ be the value returned (stored) by a memory operation. 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. $val(l_a) = val(\max_{<_m}(s_a))$ such that $\{s_a \mid s_a <_m l_a\}$ or $\{s_a \mid s_a <_p l_a\}$. 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 be postponed after a load, i.e. the $store \rightarrow load$ order relaxation. The last rule specifies that the value retrieved by a load is the one of the last store in memory order that precedes the load in memory 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 example, the following is a valid TSO memory order for the program of Table 1 that allows the program to terminate: $load(p_1, x, 1)$, $load(p_1, y, 0)$, $load(p_2, y, 1)$, $load(p_2, x, 0)$, $store(p_1, x, 1)$, $store(p_2, y, 1)$. Note that in SC, memory order has to be fully compatible with program order, and thus this memory order is not possible.

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. In this description (see Figure 1), stores from each process are buffered and eventually committed to main memory in an interleaved way. When a process executes a load, it reads the most recent value in its store buffer or, if there is none, the value present in the shared memory.

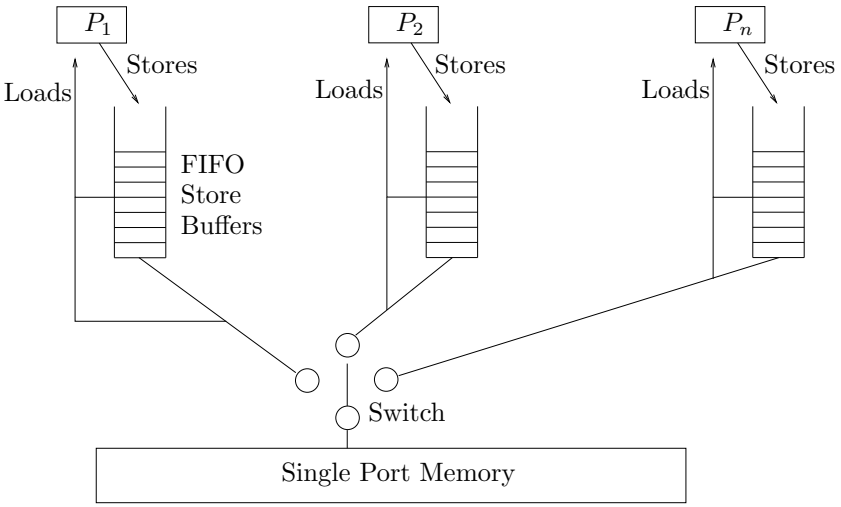

Fig. 1. Operational definition of TSO of Appendix K of [2]

This model can be formalized as follows. One introduces a set

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

of store buffers, one for each process[1]. A global state is thus composed of the content of the memory, and, for each process, a control location and a store buffer. The content $[b_p]$ of a buffer b_p is then a word in $(\mathcal{M}, \mathcal{D})^*$. The program will then execute load and store operations on these buffers. Furthermore a *commit* operations that removes the oldest store operations from a buffer and writes the corresponding value to memory can nondeterministically be executed at all times. The precise semantics of these operations can be described as follows.

store operation: $store(p, m, d)$**:**

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

load operation: $load(p, m, d)$**:**

Let $[b_p] = (m_1, d_1)(m_2, d_2)\ldots(m_f, d_f)$ and let $i = \max_{i \in \{1,\ldots,f\}}\{i \mid m_i = m\}$. If i exists, then the result of the load is the test $d_i == d$. If not, it is the result of the test $[m] == d$, where $[m]$ denotes the content of the memory location m.

commit operation: $commit(p)$**:**

Let $[b_p] = (m_1, d_1)(m_2, d_2)\ldots(m_f, d_f)$. Then, if $[b_p] \neq \varepsilon$, the result of the commit operation is

$$[b_p] \leftarrow (m_2, d_2)\ldots(m_f, d_f)$$

and

$$[m_1] \leftarrow d_1.$$

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

Finally, in programs we will also use an operation *sync* whose effect is to commit to memory the full content of all buffers.

3 Representing Sets of Buffer Contents

If store buffers are unbounded, introducing them leads to a potentially infinite state space. Furthermore, even if store buffers are bounded, they very quickly lead to unmanageably large state spaces, even for very simple programs.

To cope with this problem, we turn to the techniques that have been proposed in [15] and in [7] to represent sets of buffer contents by finite automata. In

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

this approach, sets of possible buffer contents are represented by finite automata and the state-space of the system is explored by manipulating sets of possible contents for each control location as a single object. It is clear that while exploring the state-space of a system, one can combine into a single representation the buffer contents corresponding to identical control locations. However, this will only lead to finite sets of contents being represented as a single object, whereas real gains can only come from manipulating together infinite sets of buffer contents. For achieving this, acceleration techniques are needed. Similarly to what is done in the previously cited work, we will focus on cycles in the program code and provide algorithms for directly computing the effect of iterating a sequence of operations and unbounded numbers. Before turning to this, we will first introduce the representation of sets of buffer contents by automata and see how load store and commit operations can be extrapolated to operations on automata representing sets of buffer content.

We represent the content of each buffer by a separate automaton over the alphabet $\mathcal{M} \times \mathcal{D}$ and use the following definition.

Definition 1. *A buffer automaton associated to a process p is a finite automaton $A_p = (S, \Sigma, \Delta, S_0, F)$, where*

- *S is a finite set of states,*
- *$\Sigma = \mathcal{M} \times \mathcal{D}$ is the alphabet of buffer elements,*
- *$\Delta \subseteq S \times (\Sigma \cup \{\varepsilon\}) \times S$ is the transition relation,*
- *$S_0 \subseteq S$ is a set of initial states, and*
- *F is a set of final states.*

A buffer automaton A_p represents a set of buffer contents $L(A_p)$, which is the language of the words accepted by the automaton according to the usual definition.

We have defined buffer automata to be nondeterministic, but for implementation purposes we will usually work with reduced deterministic automata. In this case, the transition relation becomes a transition function $\delta : S \times \Sigma \to S$ and the set of initial states becomes a single state s_0.

Operations on buffers can be extrapolated to operations on buffer automata as follows.

store operation: $store(p, m, d)$**:**

The result of the operation is an automaton A'_p such that

$$L(A'_p) = L(A_p) \cdot \{(m, d)\}$$

One thus simply concatenates that new stored value to all words in the language of the automaton.

load operation: $load(p, m, d)$**:**

Load operations are nondeterministic since a buffer automaton can represent several possible buffer contents. Thus it is possible that a load operation can

succeed on some represented buffer contents and fail on others. If this is the case, the load operation must lead to a state in which the set of possible buffer contents has been restricted to those on which the load operation succeeds.
For a load operation to succeed, the tested value must be found either in the store buffer or in main memory. Precisely, a load operation succeeds when at least one of the following two conditions is satisfied:

1. The language

$$L_1 = L(A_p) \cap (\Sigma^* \cdot (m, d) \cdot (\Sigma \backslash \{(y, v) \mid y \neq m \wedge v \in \mathcal{D}\})^*)$$

   is nonempty.
2. The language

$$L_2 = L(A_p) \cap (\Sigma \backslash \{(m, v) \mid v \in \mathcal{D}\})^*$$

   is nonempty and $[m] = d$.

The load operation then leads to a state with a modified store buffer automaton A'_p such that

$$L(A'_p) = L_1 \cup L_2$$

if $[m] = d$ and

$$L(A'_p) = L_1$$

otherwise. Of course, if $L_1 \cup L_2 = \emptyset$, the load operation is simply not possible.

commit operation: *commit*(p)**:**

For the commit operation, we first extract the stores that can be committed to memory. These are the stores (m, α) such that

$$(m, \alpha) \in \text{first}(L(A_p)),$$

where first(L) denotes the language of the first symbols of the words of L. Since there can be more than one such store, we need to modify the store buffer automaton according to the committed store (m, α). We have

$$L(A'_p((m, \alpha))) = \text{suffix}^1(L(A_p) \cap ((m, \alpha) \cdot \Sigma^*)),$$

where suffix$^1(L)$ denotes the language obtained by removing the first symbol of the words of L.

4 State Space Exploration and Cycle Detection

Our state-space exploration algorithm is based on a classical depth-first search. The major modification we introduce is the detection of cycles and an acceleration technique for directly computing the effect of repeatedly executing a detected cycle. The cycles we detect are those that only modify a single store

buffer. This might seem restrictive, but notice that the use of store buffers introduces a lot of independence between processes and experiments show that considering only single process cycles is sufficient. The independence induced by store buffers has however a drawback, which is that it makes the same cycles possible from many different global control locations. Proceeding naively thus results in detecting the same cycle many times over, which is unnecessary and very wasteful. To avoid this, we used the sleep set partial order reduction of [9]. This reduction avoids re-exploring transitions after executing other independent transitions. In general, the sleep set reduction does not reduce the number of states visited, but only the number of transitions followed. This is already very valuable when working with automata symbolic representations, since these increase the cost of comparing states. Furthermore, the fact that we are working with sets of states and not individual states does make sleep sets yield a reduction of the size of the state graph that needs to be explored, as we will illustrate by an example further down.

In the sleep set exploration algorithm, a set of transitions, called a sleep set, is associated with each state. Initially, the sleep set is empty. Once a transition is executed, it is added to the sleep set of the resulting state, but transitions in the sleep set that are not independent with respect to the executed transition are removed. Transitions in the sleep set associated to a state are not executed from that state. The basic depth-first search algorithm using sleep sets is given in Figure 2. We will use a crude but sufficient notion of independence. In a state s,

1. transitions of the same process are never independent;
2. transitions of different processes other than *commit* or *sync* are always independent;
3. a *commit(p)* transition of a process p is independent with the transitions of a process p', provided that, for every memory location m affected by this *commit*operation, either p' does not use m, or p' has a value for m in its store buffer, i.e., all words of the language $L(A_p)$ contain an occurrence of (m, v) for some $v \in \mathcal{D}$.
4. a *sync* operation is not independent with any other transition.

What we add to this is cycle detection and acceleration. Cycle detection is done when there is a state on the current search stack that only differs from the state being generated by the content of one store buffer. The algorithm used is the one in Figure 3.

First, we need to define when a state is included in another. A state s is included in another state s' if

1. s and s' are identical with respect to control locations and memory content, and
2. for each process p, $L(A_p(s))$ is included in $L(A_p(s'))$

Next, we need to make explicit the `cycleCondition(ss,s)` predicate. For this predicate to be true, three conditions have to be satisfied by the pair of

```
procedure DFS() {
  s = top(Stack)
  if (s ∈ H) {
    T_executed = enabled(s) \ H(s).Sleep
    s.Sleep = s.sleep ∩ H(s).Sleep
    H(s).Sleep = s.Sleep
  }
  else {
    T_executed = ∅
  }
  T = (enabled(s) \ s.Sleep ) \ T_executed

  for all t ∈ T do {
    ssucc = succ(s, t)
    ssucc.Sleep = {tt | tt ∈ s.Sleep ∧ (t, tt) independent in s}
    if (ssucc ∉ H) {
      enter ssucc in H
    }
    push ssucc onto Stack
    DFS()
  }
  pop(Stack)
}

init(Stack)
init(H)

s0 = initial state
s0.Sleep = ∅

push s0 onto Stack
insert s0 into H

DFS()
```

Fig. 2. Basic depth-first search algorithm using sleep sets

```
procedure DFS() {
  s = top(Stack)
  if (∃ sH ∈ H | s ⊆ sH) {
    T_executed = enabled(s) \ sH.Sleep
    s.Sleep = s.sleep ∩ sH.Sleep
    sH.Sleep = s.Sleep
  }
  else {
    T_executed = ∅
  }

  for all ss in (Stack \ top(Stack)){
   /* Go through stack from top to bottom */
    if (cycleCondition(ss, s)) {
      s = cycle(ss, s)
      break;
    }
  }
  T = (enabled(s) \ s.Sleep) \ T_executed
  for all t ∈ T do {
    ssucc = succ(s, t)
    ssucc.Sleep = {tt | tt ∈ s.Sleep ∧ (t, tt) independent in s}
    if (ssucc ∉ H) {
      enter ssucc in H
    }
    push ssucc onto Stack
    DFS()
  }
  pop(Stack)
}

init(Stack)
init(H)

s0 = initial state
s0.Sleep = ∅

push s0 onto Stack
insert s0 into H

DFS()
```

Fig. 3. Search algorithm with cycle detection

global states ss, s. Remembering that global states are composed of the control location of each process, the content of the memory and the buffer automata of each process, these conditions can be defined as follows.

1. s and ss are identical, except for the store buffer automaton of a single process p.
2. The languages represented by the store buffer automaton of p in ss can be extended to match the language of the store buffer automaton of p in s, i.e. there exists a word w such that $(L(A_p(s)) = L(A_p(ss)) \cdot \{w\}$.
3. The store buffer automaton obtained for s is *load equivalent* to the one of ss, i.e. the results of loads will be the same, whether starting from s or ss. Since the only difference between $L(A_p(s))$, and $L(A_p(ss))$ is the suffix w, this will be verified by checking the following condition. For all memory locations m for which there is a store operation $store(p, m, v)$ in w, let v_{last} be the value in the last store operation in w. Then the operation $load(p, m, v_{last})$ must be simultaneously possible in both s and ss and must not modify the store buffer automata $A_p(s)$ and $A_p(ss)$.

Finally, once a possible cycle content w has been detected and the conditions for a cycle are satisfied, we need to store the buffer automaton representing the buffer contents that can be obtained by repeating the cycle. This will simply be the automaton $A_p^{cycle}(ss)$ accepting $L(A_p(ss)) \cdot \{w\}^*$. Thus the operation `cycle`(ss, s) of our search algorithm simply replaces the store buffer automaton for process p in state s by the automaton $A_p^{cycle}(ss)$.

Example 1. We illustrate the state-space reduction that can be obtained by the use of sleep sets. Figure 4 shows the control graph of two processes p_0 and p_1. In Figure 5, part of the global state graph of this system is shown. In state 4, a cycle has been detected for the store buffer of p_0, yielding

$$(x, 1)((x, 0)(x, 1))^*$$

as set of possible contents. In state 6, the buffer has become

$$(x, 1)(x, 0)(x, 1)((x, 0)(x, 1))^*,$$

and thus, state 6 is included in state 4. In state 5, if we don't add the transition $st(p_0, x, 1)$ (which led to state 6) to the sleep set of state 7, we will end up generating states 8 and 9 before detecting any state inclusion and add many more states to the search graph.

5 Experimental Results

We have implemented our method in a prototype tool. This tool takes as input a slightly modified Promela model. The prototype has been implemented in Java, and uses the BRICS automata-package [8] to handle our store-buffers.

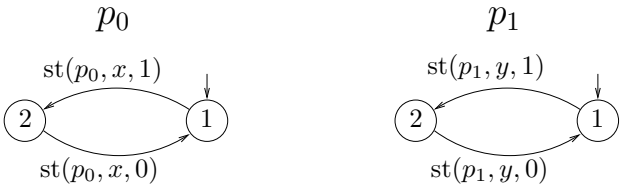

Fig. 4. Control graphs of two processes p_0 and p_1

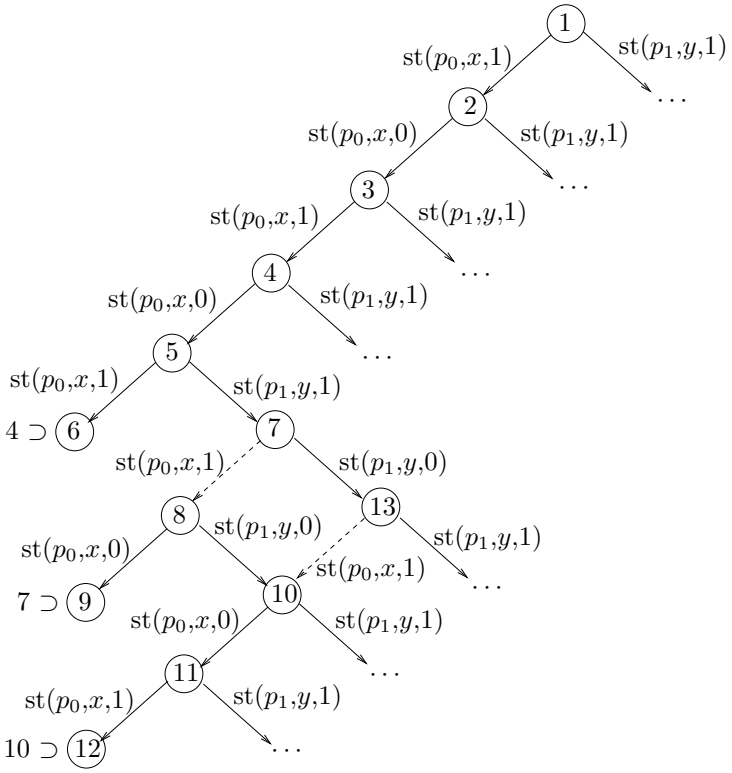

Fig. 5. Global exploration graph showing reduction using sleep sets

We have tested our implementation on several programs and protocols. One of our test programs is a program (see figure 6)[2] where the first process may cycle indefinitely from the initial state, but where the second depends on the global memory being modified to be able to move. Indeed, P_1 can directly cycle indefinitely, writing the infinite sequence $(x,1)(x,0)(x,1)(x,0)(x,1)(x,0)\ldots$ to its store buffer. This cycle is detected and all possible contents of P_1's store buffer represented. Then, the process P_2 can, once the cycle in the buffer of P_1 is established, "consume" this cycle, which unlocks its own cycle. For example, a global state such as $(1,1,0,0,((x,1)(x,0))^*,((y,1)(y,0))^*)$ (where the notation is (p_1, p_2, m_1, m_2, b_1, b_2)) will eventually be reached. Consuming means that store operations are committed to the global memory, without that the process itself moving.

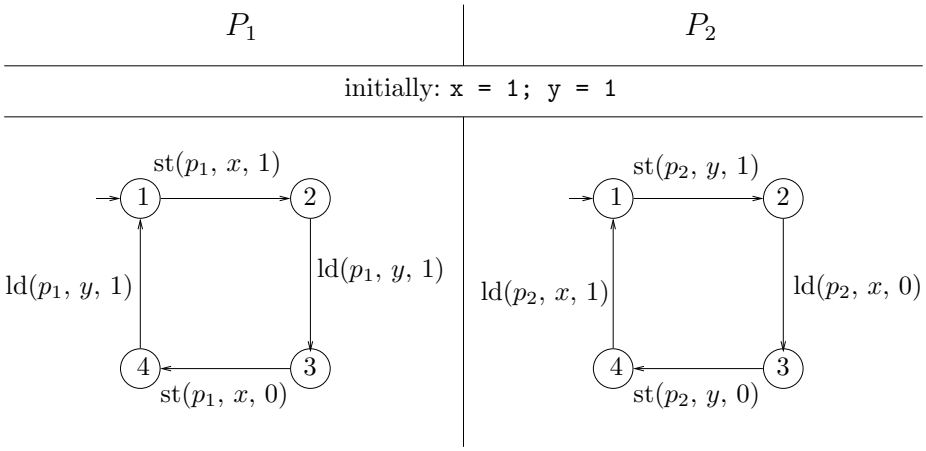

Fig. 6. Example program unlocking a cycle

Moreover, under SC, if both processes are in state 4, the program is in a deadlock. In TSO, there is the possibility of deadlock, but it is also possible that the program may continue (if there are buffered store operations), and thus the values of x and y may change value. Interestingly, both behaviors have also been observed while running a C implementation of this program on a dual core processor.

Another classical algorithm often analyzed in the context of relaxed memory models is Peterson's algorithm for mutual exclusion. We have considered single entry and repeated entry versions of this algorithm. In the single entry version, the two processes want to enter into the critical section only one time. Verifying this can be done with our implementation, as well as with other tools, such as those of [6], [5] or [16]. Verification becomes more difficult when considering the

[2] For readability, the operations *store* and *load* are changed in *st* and *ld*.

repeated entry version. In this version, both processes want to enter the critical section an arbitrary number of times. Using our prototype, we could finish the exploration within 90 seconds, finding the error, or, when adding some memory fences at the right places, showing the absence of errors.

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. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-tso. In Berghofer, S., Nipkow, T., Urban, C., Wenzel, M., eds.: TPHOLs. Volume 5674 of Lecture Notes in Computer Science., Springer (2009) 391–407
5. Mador-Haim, S., Alur, R., Martin, M.: Plug and play components for the exploration of memory consistency models. Technical report, University of Pennsylvania (2010)
6. 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.: PLDI, ACM (2007) 12–21
7. Boigelot, B., Godefroid, P., Willems, B., Wolper, P.: The power of QDDs (extended abstract). In Hentenryck, P.V., ed.: SAS. Volume 1302 of Lecture Notes in Computer Science., Springer (1997) 172–186
8. Møller, A.: brics/automaton DFA/NFA Java implementation.
9. 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)
10. Park, S., Dill, D.L.: An executable specification, analyzer and verifier for rmo (relaxed memory order). In: SPAA. (1995) 34–41
11. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In Gupta, A., Malik, S., eds.: CAV. Volume 5123 of Lecture Notes in Computer Science., Springer (2008) 107–120
12. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In Hermenegildo, M.V., Palsberg, J., eds.: POPL, ACM (2010) 7–18
13. Intel Corporation: Intel®64 and IA-32 Architectures Software Developer's Manual. Specification (2007) http://www.intel.com/products/processor/manuals/index.htm.
14. Loewenstein, P., Chaudhry, S., Cypher, R., Manovit, C.: Multiprocessor memory model verification. (2008)
15. Boigelot, B., Wolper, P.: Symbolic verification with periodic sets. In Dill, D.L., ed.: CAV. Volume 818 of Lecture Notes in Computer Science., Springer (1994) 55–67
16. Hangal, S., Vahia, D., Manovit, C., Lu, J.Y.J., Narayanan, S.: Tsotool: A program for verifying memory systems using the memory consistency model. In: ISCA, IEEE Computer Society (2004) 114–123