TSO to SC via Symbolic Execution

Heike Wehrheim and Oleg Travkin(✉)

Institut für Informatik, Universität Paderborn, 33098 Paderborn, Germany
{wehrheim,oleg82}@uni-paderborn.de

Abstract. Modern multi-core processors equipped with weak memory models like TSO exhibit executions which – due to store buffers – seemingly reorder program operations. Thus, they deviate from the commonly assumed *sequential consistency* (SC) semantics. Analysis techniques for concurrent programs consequently need to take reorderings into account. For TSO, this is often accomplished by explicitly modelling store buffers.

In this paper, we present an approach for reducing TSO-verification of concurrent programs (with fenced or write-free loops) to SC-verification, thereby being able to reuse standard verification tools. To this end, we transform a given program P into a new program P' whose SC-semantics is (bisimulation-) equivalent to the TSO-semantics of P. The transformation proceeds via a symbolic execution of P, however, only with respect to store buffer contents. Out of the thus obtained abstraction of P, we generate the SC program P' which can then be the target of standard analysis tools.

1 Introduction

With the advent of multi-core processors we recently see new types of bugs in concurrent programs coming up[1]. These bugs are due to the weak memory semantics of multi-core processors, which in their architectures are streamlined towards high performance. In executions of concurrent programs, weak memory causes program statements to seemingly be executed in an order different from the given program order. TSO (total store order) is one such weak memory model (of the x86 processors [18]), incorporating characteristics common to a lot of other weak memory models. On the contrary, concurrent executions adhering to program order are said to be *sequentially consistent* (SC) [14].

As concurrent programs are today executed on multi-cores, analysis techniques for concurrent software need to be based on weak memory semantics. This is often accomplished by an explicit modelling of store buffers, which are the cause of statement reordering. Store buffers are attached to cores, and values of variables shared among processes are first written to the corresponding store buffer before being flushed to main memory. Thereby, a read operation following a write may seem to overtake it from the point of view of other process. Analysis techniques employing store buffer modelling are for instance model checking

[1] See e.g. T. Lane. Yes, waitlatch is vulnerable to weak-memory-ordering bugs, http://www.postgresql.org/message-id/24241.1312739269@sss.pgh.pa.us, 2011.

© Springer International Publishing Switzerland 2015
N. Piterman (Ed.): HVC 2015, LNCS 9434, pp. 104–119, 2015.
DOI: 10.1007/978-3-319-26287-1_7

approaches [19,23], predicate abstraction [11], or interactive proving [20]. The modelling of store buffers does, however, impose a non-neglectable overhead on the analysis, might it be automatic, supported by theorem provers or manual.

In this paper, we propose an approach for reducing TSO analysis (i.e., analysis of concurrent programs taking the TSO semantics into account) to SC analysis. The technique is applicable to all programs with *fenced* or *write-free* loops, i.e., programs in which all loops contain at least one fence operation (memory barrier), or alternatively have no write operations in loops. While this seems rather restrictive, a lot of concurrent algorithms already possess this property, e.g., concurrent data structures using compare-and-swap operations (acting as fences) in loop conditions. Our approach proceeds by translating a program P into a program P' such that P's TSO semantics is bisimilar to P''s SC semantics. Like [5], the approach closest to ours, the additional condition (fenced or write-free loops) guarantee finite (though a priori unknown) store buffer sizes during execution. Unlike [5], we however only need few additional program variables in the constructed program P', and these are furthermore all local to processes. As a result, our technique is *compositional* in that it separately translates the programs of processes in a parallel composition.

The translation proceeds via a sort of symbolic execution of P [17] which constructs an abstraction of P symbolically tracking store buffer contents. This abstraction is transformed into a new program P' (by an approach for program generation out of abstract reachability graphs [22]). For the generated program we can afterwards re-use established analysis techniques and tools for SC. To show the practicability of our approach, we apply the technique to four mutual exclusion algorithms and two concurrent data structures, which we translate into an SC version and give to the model checker SPIN [13]. In almost all cases, it can be seen that the number of states generated by SPIN is reduced when going from a TSO version with explicit store buffer modelling to our SC version.

The paper is structured as follows. We start with a short introduction to weak memory models and the reorderings they generate. We will then proceed with defining the syntax and semantics of single processes, both for TSO and SC. On the semantic domain, we define our notion of equivalence (of programs viz. their executions). Sections 4 and 5 explain the transformation of a program into an SC form. We report on experimental results in Sect. 6 and discuss related work in Sect. 7. Section 8 concludes.

2 Weak Memory Reorderings

Weak memory models describe the semantics of concurrent programs when executed on multi-core machines. In general, the execution of memory instructions on the TSO memory model involves the usage of store buffers local to processes. A write operation on a shared variable thus first puts its written value into the store buffer. The contents of store buffers are occasionally *flushed* into main memory. Memory barriers (fences) can be used to enforce flushing, because fence operations can only be executed when the store buffer is empty. All read operations on shared variables will first examine the contents of the process' store

buffer: if there is a value for the variable in the store buffer, the read will take the most recent one, otherwise it reads from main memory.

This usage of store-buffers leads to two kinds of reorderings on TSO: write-read reorderings and early-reads. These effects can best be understood on examples. Such small examples exhibiting certain interesting behaviours of multi-processors are known as *litmus tests*. Figures 1 and 2 give two such litmus tests. The first example is a write-read reordering. Both processes first write on a shared variable and then read from another shared variable into local registers. Since both writes might first be placed into the local store buffers, both reads can still see the initial values of x and y, and hence $r1 = 0 \wedge r2 = 0$ is a possible final state. It looks as though the writes and reads have changed position.

Initially : $x = 0 \wedge y = 0$

Process 1	Process 2
$1 : write(x, 1);$	$1 : write(y, 1);$
$2 : read(y, r1);$	$2 : read(x, r2);$
$3 :$	$3 :$

$r1 = 0 \wedge r2 = 0$ possible

Fig. 1. Litmus test for write-read reordering

Initially : $x = 0 \wedge y = 0$

Process 1	Process 2
$1 : write(x, 1);$	$1 : write(y, 1);$
$2 : read(x, r1);$	$2 : read(y, r3);$
$3 : read(y, r2);$	$3 : read(x, r4);$
$4 :$	$4 :$

$r1 = r3 = 1 \wedge r2 = r4 = 0$ possible

Fig. 2. Litmus test for early reads

The litmus test in Fig. 2 exemplifies the phenomenon of early reads (or inter-processor forwarding). The possible outcome $r1 = 1 \wedge r2 = 0 \wedge r3 = 1 \wedge r4 = 0$ occurs when both reads from lines 2 read from the process' own written value in the store buffer, and thus at the end the processes observe different orders of writes. It looks as though the reads on lines 3 have happened before both writes.

The basic idea of our approach is now to make these reorderings explicit in a new version of the program, and give this new version to standard SC verification tools for analysis.

3 Processes and Their Parallel Composition

Reorderings are being made explicit by transforming – in a concurrent program $[P_1||\ldots||P_n]$ – the programs P_i of single processes into a form P_i' such that the following holds: execution of P_i on TSO is "equivalent" to execution of P_i' on SC. We can then use standard SC verification tools for checking properties on $[P_1'||\ldots||P_n']$. In this section, we will first of all explain how programs look like, and what we mean by "equivalent". The equivalence should in particular guarantee that other processes running in parallel cannot distinguish equivalent programs. We will thus base our equivalence on a notion of bisimulation [16].

As a consequence, our transformation need not be defined on the whole parallel program, but can transform programs of single processes in isolation: our technique is *compositional*.

For programs, we assume a set Reg of registers local to processes and a set of variables Var, shared by processes. For simplicity, both take as values just natural numbers. Thus the local state of a program is - among others - represented by a function $reg : Reg \to \mathbb{N}$, and the global state by a function $mem : Var \to \mathbb{N}$. We use the notation $mem[x \mapsto n]$ to stand for the function mem' which agrees with mem up to x which is mapped to n (and similar for other functions). Processes use local *store buffers*, i.e., FIFO queues, into which values for shared variables are first written before being flushed to main memory. A store buffer $sb \in (Var \times \mathbb{N})^*$ is a sequence of (variable and value) pairs. We write $x \in sb$ to state that there is a pair $(x, \cdot)$ in the store buffer sb. Program statements are labelled with *locations* out of some set L. A variable pc taking values $\ell \in L$ determines the statements that can be executed next. Thus, the local state s of some process is characterised by a tuple (ℓ, reg, sb).

We describe the programs of processes by *predicates*, one predicate stating conditions on the initial state and a set of predicates for operations (indexed by $i \in I$). Thus a *program* P of a process is given as $(Init, (COp^i)_{i \in I})$, where $Init$ is a predicate on pc, reg and sb. Each COp^i is a predicate over pc, pc', reg, reg', sb, sb' and mem, mem' in which the primed versions refer to the state after executing the operation. We assume $Init$ to specify $sb = \langle\,\rangle$ (empty sequence).

Processes are allowed to interact with the memory using pre-defined operations from the set $\{write, read, fence, flush\}$. The latter two are only meaningful in a TSO context. In TSO, their semantics is defined by the following predicates.

Write: Writing the value of a register or a constant to a variable x:
$write(x, n) \mathrel{\widehat{=}} sb' = sb \frown \langle (x, n) \rangle$, writing constant n
$write(x, r) \mathrel{\widehat{=}} sb' = sb \frown \langle (x, reg(r)) \rangle$, writing value from register r

Read: Reading the value of a variable x into register r:

$$read(x, r) \mathrel{\widehat{=}} (x \notin sb \wedge read_{mem}(x, r)) \vee (x \in sb \wedge read_{loc}(x, r)) \text{ where}$$
$$read_{mem}(x, r) \mathrel{\widehat{=}} reg' = reg[r \mapsto mem(x)]$$
$$read_{loc}(x, r) \mathrel{\widehat{=}} reg' = reg[r \mapsto latest(x, sb)]$$
$$latest(x, sb) = n \mathrel{\widehat{=}} \exists sb_{pre}, sb_{suf} : sb = sb_{pre} \frown \langle (x, n) \rangle \frown sb_{suf} \wedge x \notin sb_{suf}$$

Fence: Memory barrier blocking until store buffer is empty:
$fence \mathrel{\widehat{=}} sb = \langle\,\rangle$

Flush: Flushing single store buffer entries to main memory:
$flush \mathrel{\widehat{=}} \exists (x, v) \in (Var \times \mathbb{N}) : sb = \langle (x, v) \rangle \frown sb' \wedge mem' = mem[x \mapsto v]$

The operation $read$ has two cases: Reads might be *early*, reading from the contents of the store buffer ($read_{loc}$) or - if the store buffer contains no entry for the variable - read from main memory ($read_{mem}$). In addition to the above operations we have thread-local operations $LocOp$, i.e., operations of the form $r := expr$ (semantics $reg' = reg[r \mapsto reg(expr)]$), where $expr$ is an expression

build out of constants and register names using e.g. arithmetic operations, or boolean conditions over registers and constants.

We assume all operations (predicates) to have the following form (or equivalent):

$$COp \mathrel{\widehat{=}} pc = \ell \wedge pc' = \ell' \wedge op$$

where op is either a memory operation or a local operation, e.g., $op = read(x, r)$.

We define $op(COp) \mathrel{\widehat{=}} op$ to state the operational part of COp (similar for $Init$), without the part referring to program locations, and $pc(COp)$ to be the part of the predicate referring to the program counter. When executed on TSO, we implicitly add the operation $COp^f \mathrel{\widehat{=}} flush$ to each process. This is the only operation without location predicates. We let $def(op)$ be the set of registers assigned to (changed) in an operation op and $use(op)$ to be the registers used. Process 1 of Fig. 1 would be specified as follows:

$$\begin{aligned} Init &\mathrel{\widehat{=}} pc = 1 \wedge r1 = 0 \wedge sb = \langle\,\rangle \\ COp^1 &\mathrel{\widehat{=}} pc = 1 \wedge pc' = 2 \wedge write(x, 1) \\ COp^2 &\mathrel{\widehat{=}} pc = 2 \wedge pc' = 3 \wedge read(y, r1) \\ COp^f &\mathrel{\widehat{=}} flush \end{aligned}$$

The semantics of programs is given by labelled transition systems. For being able to compare the semantics of programs run under TSO with those run on SC, we define a *common* set of labels:

$$\begin{aligned} Lab = \{&skip, r := expr, bexpr, wr(x, n), rd(x, r) \mid \\ &x \text{ variable}, r \text{ register}, n \in \mathbb{N}\} \end{aligned}$$

For the semantics, we use the convention that all entities (registers, pc, ...) which are not mentioned in the operation formula keep their values. For pairs of (global) states (g, g'), $g = (\ell, reg, sb, mem)$, $g' = (\ell', reg', sb', mem')$ we write $(g, g') \models COp$ to say that the predicate COp is satisfied by states g and g'. Similarly, for predicates p on unprimed variables only and states g, we write $g \models p$ to say that the predicate p is valid in the state g.

For single processes, we next define an *open* semantics. It is open in the sense that we assume other processes, possibly running in parallel, to arbitrary change shared memory, and thus incorporate into the semantics all steps the process can do with *arbitrary* values of mem.

Definition 1. *The* process-local TSO transition system of a program $P = (Init, (COp^i)_{i \in I})$, $[\![P]\!]_{tso}$, *is* $(S, \longrightarrow, S_0)$ *with*

- $S_0 = \{s = (\ell, reg, sb) \mid s \models Init\}$,
- $s \xrightarrow{lab} s'$ *with* $s = (\ell, reg, sb)$ *and* $s' = (\ell', reg', sb')$ *iff* $\exists\, COp^i, \exists\, mem, mem'$ *s.t.* $((mem, s), (mem', s')) \models COp^i$, *and the label* lab *is*
  - $r := reg(expr)$ *if* $op(COp^i) = (r := expr)$,
  - $reg(bexpr)$ *if* $op(COp^i) = bexpr$,

- *skip if* $op(COp^i) \in \{fence, write(x,n), write(x,r), skip\}$,
- $wr(x,n)$ *if* $op(COp^i) = flush$ *and* $sb = \langle (x,n) \rangle \frown sb'$,
- $rd(x,r)$ *if* $op(COp^i) = read(x,r)$ *and* $x \notin sb$, *and*
- $r := n$ *if* $op(COp^i) = read(x,r)$, $x \in sb$ *and* $n = latest(x, sb)$.

For such transitions we use the notation $s \xrightarrow{lab}_{mem,mem'} s'$.

– *S is the set of all states reachable from* S_0 *by transitions.*

The choice of labels reflects what is visible to the environment (i.e., other processes): a local write to a store buffer looks to the outside as if nothing happens, hence gets a *skip* label; a local read from store buffer looks like an assignment to a register, and hence gets an assignment label; finally, a flush operation looks to the outside like a proper write on shared memory and thus is labelled as write. This idea of *relabelling* transitions according to what effects are visible to the outside is also the basic principle of our TSO to SC transformation.

For the SC semantics of programs, we slightly restrict the set of operations. In SC, programs cannot (and do not) have fence operations, and furthermore their write and read predicates have a different semantics.

Write: Writing the value of a register or a constant to a variable x:
$write_{sc}(x,n) \mathrel{\widehat{=}} mem' = mem[x \mapsto n]$, writing constant n
$write_{sc}(x,r) \mathrel{\widehat{=}} mem' = mem[x \mapsto reg(r)]$, writing value from register r
Read: Reading the value of a variable x into register r:
$read_{sc}(x,r) \mathrel{\widehat{=}} reg' = reg[r \mapsto mem(x)]$

As we see now, none of the operation predicates is refering to the store buffer. The local states of the SC transition system are thus of the form (ℓ, reg). Note that in this case we do not implicity add a flush operation to the set of program operations.

Definition 2. *The* process-local SC transition system of a program $P = (Init, (COp^i)_{i \in I})$, $[\![P]\!]_{sc}$, *is* $(Q, \rightarrow, Q_0)$ *with* $Q_0 = \{q = (\ell, reg) \mid q \models Init\}$ *and* $q \xrightarrow{lab} q'$ *with* $q = (\ell, reg)$ *and* $q' = (\ell', reg')$ *iff* $\exists COp^i, \exists mem, mem' :$ $((mem, q), (mem', q')) \models COp^i$ *and the label lab is*

– $r := reg(expr)$ *if* $op(COp^i) = (r := expr)$,
– $reg(bexpr)$ *if* $op(COp^i) = bexpr$,
– $skip$ *if* $op(COp^i) = skip$,
– $wr(x,n)$ *if* $op(COp^i) \in \{write_{sc}(x,n), write_{sc}(x,r)$ *and* $reg(r) = n\}$,
– $rd(x,r)$ *if* $op(COp^i) = read_{sc}(x,r)$.

Again, Q is the set of all reachable states.

Processes typically run in parallel with other processes. The semantics for parallel compositions of processes is now a *closed* semantics already incorporating all relevant components. We just define it for two processes here; a generalisation to larger numbers of components is straightforward.

Definition 3. *Let* $P_j = (Init_j, (COp_j^i)_{i \in I})$, $j \in \{1,2\}$, *be two processes, Init an additional predicate on mem, and let* $(S_j, \rightarrow_j, S_{0,j})$, *be their process local (i.e., open) semantics (TSO or SC).*

The closed TSO or SC semantics, *respectively, of* $P_1 ||_{Init} P_2$ *is the labelled transition system* $(S, \rightarrow, S_0)$ *with* $S \subseteq \{(mem, s_1, s_2) \mid s_1 \in S_1, s_2 \in S_2\}$, $S_0 = \{s \in S \mid s \models Init_1 \wedge Init_2 \wedge Init\}$, *and* $s = (mem, s_1, s_2) \xrightarrow{lab} s' = (mem', s_1', s_2')$ *when* $(s_1 \xrightarrow{lab}_{mem,mem'} s_1' \wedge s_2 = s_2')$ *or* $(s_2 \xrightarrow{lab}_{mem,mem'} s_2' \wedge s_1 = s_1')$.

Due to the open semantics for processes, we have thus been able to give a *compositional* semantics for parallel composition.

Ultimately, we will be interested in comparing the TSO semantics of one program with the SC semantics of another. Our notion of equality is based on bisimulation equivalence [16]. Our definition of bisimulation compares transition systems with respect to their *labels* on transitions as well as their local *states*.

Definition 4. *Let* $T_1 = (S, \rightarrow_{tso}, S_0)$ *be a TSO and* $T_2 = (Q, \rightarrow_{sc}, Q_0)$ *an SC transition system.*

Transition systems T_1 *and* T_2 *are* locally bisimilar, $T_1 \approx_\ell T_2$, *if there is a bisimulation relation* $\mathcal{R} \subseteq S \times Q$ *such that the following holds:*

1. *Local state equality:*
   $\forall (s,q) \in \mathcal{R}, s = (\ell_1, reg_1, sb), q = (\ell_2, reg_2), \forall r \in Reg$: $reg_1(r) = reg_2(r)$.
2. *Matching on initial states:*
   $\forall s_0 \in S \, \exists \, q_0 \in Q_0$ *s.t.* $(s_0, q_0) \in \mathcal{R}$, *and reversely* $\forall \, q_0 \in Q_0 \, \exists \, s_0 \in S_0$ *s.t.* $(s_0, q_0) \in \mathcal{R}$.
3. *Mutual simulation of steps:*
   if $(s_1, q_1) \in \mathcal{R}$ *and* $s_1 \xrightarrow{lab}_{tso} s_2$ *then* $\exists \, q_2$ *such that* $q_1 \xrightarrow{lab}_{sc} q_2$ *and* $(s_2, q_2) \in \mathcal{R}$, *and reversely, if* $(s_1, q_1) \in \mathcal{R}$ *and* $q_1 \xrightarrow{lab}_{sc} q_2$ *then* $\exists \, s_2$ *such that* $s_1 \xrightarrow{lab}_{tso} s_2$ *and* $(s_2, q_2) \in \mathcal{R}$.

Similarly, one can define *global bisimilarity* for the closed semantics of a parallel composition, in addition requiring equality of shared memory *mem*. We use the notation $\approx_g$ to denote global bisimilarity. This lets us state our first result: Local bisimilarity of processes implies global bisimilarity of their parallel compositions.

Theorem 1. *Let* P_1, P_1', P_2, P_2' *be processes such that* $[\![P_1]\!]_{tso} \approx_\ell [\![P_1']\!]_{sc}$ *and* $[\![P_2]\!]_{tso} \approx_l [\![P_2']\!]_{sc}$ *and let Init be a predicate on mem. Then*

$$[\![P_1 || _{Init} P_2]\!]_{tso} \approx_g [\![P_1' || _{Init} P_2']\!]_{sc}.$$

Proof idea: Let $\mathcal{R}_i$ be the bisimulation relations showing $[\![P_i]\!]_{tso} \approx_\ell [\![P_i']\!]_{sc}$. Then

$$\mathcal{R} := \{((mem, s_1, s_2), (mem, s_1', s_2')) \mid (s_1, s_1') \in \mathcal{R}_1 \wedge (s_2, s_2') \in \mathcal{R}_2\}$$

is the relation showing global bisimilarity. □

This result enables us to carry out the transformation from TSO to SC *locally*, i.e., transform the programs of processes individually and after that combine their SC versions in parallel.

4 Symbolic Store-Buffer Graphs

The basic principle behind our verification technique is to transform every program P into a program P' such that $[\![P]\!]_{tso}$ is locally bisimilar to $[\![P']\!]_{sc}$. The construction of P' proceeds by symbolic execution of P and out of the thus constructed symbolic states generation of P'. The symbolic execution tracks - besides the operations being executed and the program locations reached - store buffer contents *only*, and only in a symbolic form. The symbolic form stores variable names together with either values of $\mathbb{N}$ (in case a constant was used in the *write*), or register *names* (in case a register was used). A symbolic store buffer content might thus for instance look like this: $\langle (x,3),(y,r_1),(x,r_2),(z,5)\rangle$. The symbolic execution thereby generates a symbolic reachability graph, called store-buffer graph.

Definition 5. *A* store-buffer (or sb-)graph $G = (V, E, v_0)$ *consists of a set of nodes* $V \subseteq (L \times (Var \times (Reg \cup \mathbb{N}))^*)$*, edges* $E \subseteq V \times Lab_{tso} \times V$ *and initial node* $v_0 \in V$ *where* $Lab_{tso} = \{write(x,r), write(x,n), read(x,r), flush, fence\} \cup LocOp$.

The store-buffer graph for a program P is constructed by a form of symbolic execution, executing program operations step by step without constructing the concrete states of registers. We let $tail(list)$ of a nonempty sequence $list$ denote the sequence without its first element.

Definition 6. *Let* $P = (Init, (COp^i)_{i\in I})$ *be the program of a process. The* sb-graph *of* P*,* $sg(P)$*, is inductively defined as follows:*

1. $v_0 := (\ell_0, \langle\,\rangle)$ *with* $Init \Rightarrow pc = \ell_0$,
2. *if* $(\ell, ssb) \in V$*, we add a node* (ℓ', ssb') *and*
   - *an edge* $(\ell, ssb) \xrightarrow{lab} (\ell', ssb')$ *if* $\exists\, COp^i$ *with* $op(COp^i) = lab$ *and*
     - $lab = flush$, $\ell = \ell'$ *and* $ssb \neq \langle\,\rangle$ *and* $ssb' = tail(ssb)$*, or* $pc(COp^i) = (pc = \ell \wedge pc' = \ell')$ *and*
       - $lab \in LocOp$ *and* $ssb' = ssb$*, or*
       - $lab = write(x,r)$ *and* $ssb' = ssb \frown \langle (x,r)\rangle$*, or*
       - $lab = write(x,n)$ *and* $ssb' = ssb \frown \langle (x,n)\rangle$*, or*
       - $lab = fence$ *and* $ssb = ssb' = \langle\,\rangle$*, or*
   - *an edge* $(\ell, ssb) \xrightarrow{read_{mem}(x,r)} (\ell', ssb')$ *and a node* (ℓ', ssb') *if* $\exists\, COp^i = (read(x,r) \wedge pc = \ell \wedge pc' = \ell')$ *and* $ssb' = ssb \wedge x \notin ssb$*, or*
   - *an edge* $(\ell, ssb) \xrightarrow{read_{loc}(x,r)} (\ell', ssb')$ *and a node* (ℓ', ssb') *if* $\exists\, COp^i = (read(x,r) \wedge pc = \ell \wedge pc' = \ell')$ *and* $ssb' = ssb \wedge x \in ssb$.

As an example, Figs. 3 and 4 show the store-buffer graphs of process 1 from Fig. 1 and of process 1 in Fig. 2, respectively. On them, we directly see when the effects of writes take place in main memory, namely when the corresponding flush happens. On the left graph, right branch, we thus see the read of y happening before the "real" write of x (flush) to memory. On the right graph, right branch, we see the read of x taking place before the write to x (flush). Later we will

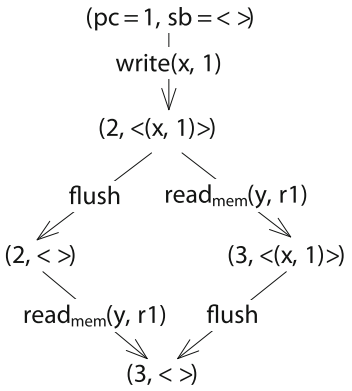

Fig. 3. Store buffer graph representing the reachable store buffer states of process 1 in Fig. 1.

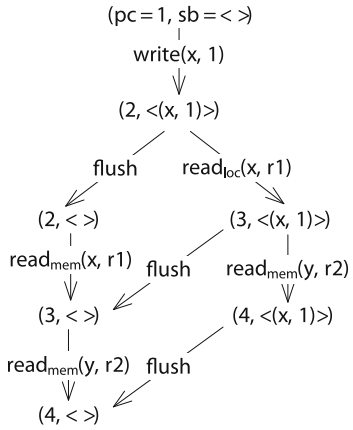

Fig. 4. Store buffer graph representing the reachable store buffer states of process 2 in Fig. 2.

see that all such early reads still read correct values in the SC version of the program.

Note that store-buffer graphs need not necessarily be finite. They are infinite if a program has loops with write operations, but no fences in order to enforce flushing of store buffer content. Since this finiteness of the store buffer graph is key to our technique, we next define our only restriction on the class of programs considered: all loops have to be *fenced* or *write-free*.

We first define loops. A (syntactically possible) path of a program P is a sequence $\ell_1, \ell_2, ..., \ell_n$ of locations such that there are operations $COp^1, ..., COp^{n-1}$ such that $pc(COp^i) = (pc = \ell_i \wedge pc' = \ell_{i+1})$. We also write paths like this: $\ell_1 \xrightarrow{COp^1} \ell_2 \xrightarrow{COp^2} \dots \xrightarrow{COp^{n-1}} \ell_n$. A *loop* is a path $\ell_1, \ell_2, ..., \ell_n$ such that $n > 1$ and $\ell_1 = \ell_n$. A loop is *write-free* if none of the operations on the loop is a write. A loop is *fenced*, if at least one of the operations on the loop is a fence. We furthermore assume that all process programs are in *SSA-form* (static single assignment [10]), meaning that all the registers are (statically) assigned to only once, i.e., for every register r there is at most one operation op with $r \in def(op)$. We furthermore assume that registers are never used before defined. Both, this and the SSA-form is guaranteed by modern compilers, e.g., the LLVM-framework[2] which we use for our approach only generates intermediate code in this form.

Proposition 1. *Let P be a process program in which every loop is fenced or write-free. Then $sg(P)$ is finite.*

[2] http://www.llvm.org.

In the generation of a new program out of an sb-graph we transform every edge of the graph into an operation (predicate). In this, a flush operation in the sb-graph, flushing a symbolic store buffer content (x, r) (r being a register name), becomes a $write_{sc}(x, r)$ operation. For this to be sound (w.r.t. the intended equivalence of old and new program), we need to make sure that the content of register r at a flush is still the same as the one at the time of writing the pair (x, r) into the (symbolic) store buffer. This is not necessarily the case. A path $\ell_1 \xrightarrow{COp^1} \ell_2 \xrightarrow{COp^2} \ldots \xrightarrow{COp^{n-1}} \ell_n$ is a *write-def* chain (wd-chain) if there is an $r \in Reg$ such that $op(COp^1) = write(., r)$ and $r \in def(COp^{n-1})$. A wd-chain is *fenced*, if one of the operations in between the write and the definition of the register is a fence. If this is guaranteed, we know that a register occuring with its name in the symbolic store buffer still has the same value as of the corresponding write.

Proposition 2. *Let P be the program of a process in SSA form with fenced write-def chains only. Let $s = (\ell, reg, sb)$ be a state of $[\![P]\!]_{tso}$ such that sb contains an entry $(x, n), n \in \mathbb{N}$. If this value has been put into the store buffer by an operation $write(x, r), r \in Reg$, then $reg(r) = n$.*

Proof: By the definition of wd-chains: after $write(x, r)$ there is no further operation defining r before the next fence operation. A fence, however, needs an empty store buffer in order to execute. □

As this property is key to our transformation, we next define a way of changing every program into an equivalent one with fenced wd-chains only. We first determine all write operations causing unfenced wd-chains, e.g. by a simple dataflow analysis. Let W be the set of all such *write sources* of wd-chains, and $Rg(W) \subseteq Reg$ be the set of registers participating in such writes. For every register $r \in Rg(W)$ we now introduce a new register r_{aux} and add it to Reg (thereby giving a set Reg_{aux}). Note that this is the only point in our program transformation where new variables or registers are introduced. The registers r_{aux} act as auxiliary variables in the programs. Every write $COp \in W, COp = (pc = \ell \wedge write(x, r) \wedge pc' = \ell)$ is now transformed into a new operation $COp_{aux} \mathrel{\widehat{=}} (pc = \ell \wedge write_{aux}(x, r) \wedge pc' = \ell)$ where

$$write_{aux}(x, r) \mathrel{\widehat{=}} write(x, r) \wedge reg' = reg[r_{aux} \mapsto reg(r)]$$

We let P' denote the program P with all such changes.

Proposition 3. *The program P' has no unfenced wd-chains.*

The label of this new operation $write_{aux}$ in the TSO semantics is $r_{aux} := n$ for $n = reg(r)$ (see Definition 1). Note that the number of new registers needed is bounded by the number of registers used in loops. For each COp_{aux} operation, we add an edge $(\ell, ssb) \xrightarrow{lab} (\ell', ssb')$ to the sb-graph, where $lab = write(x, r_{aux}) \wedge reg' = reg[r_{aux} \mapsto reg(r)]$ and $ssb' = ssb \frown \langle (x, r_{aux}) \rangle$. Note that we use r_{aux} in the symbolic store buffer, although the value of r is used in the transition system.

5 SC Program Generation

The store-buffer graph presents an *abstraction* of the actual TSO transition system of a program. The basic idea behind the generation of new programs out of store-buffer graphs is now to take the store-buffer graph as the control flow graph of the new program. We do so by using the nodes in the sb-graph as new program locations, i.e., program locations become pairs of (location, symbolic store buffer contents). If we would simply take the operations on edges as they are, we would arrive at a new program P' which is equivalent to P w.r.t. the TSO semantics. However, instead of using the operations as they are written on the edges, we make changes analogous to the relabelling used in the TSO semantics: The generated operation in the SC program should reflect the visible effect of an TSO operation, e.g. flush operations become writes and local reads become local assignments. For the latter, we use the fact that the symbolic store buffer contents contains *names* of registers, not just their values.

Definition 7. *Let $G = (V, E, v_0)$ be an sb-graph of a program P with init predicate $Init$. The new SC program of G, $prog(G)$, $(Init', (COp^i)_{i \in I'})$ is defined as follows:*

- *We use $Init' \mathrel{\widehat{=}} pc = v_0 \wedge op(Init)$,*
- *for every edge $v \xrightarrow{lab} v'$, we define an operation $COp^i \mathrel{\widehat{=}} pc = v \wedge pc' = v' \wedge op^{sc}(lab)$, where op^{sc} maps the edges of the sb-graph to the behaviorally equivalent steps in an SC setting:*

$$
\begin{aligned}
op^{sc}(lab) &\mathrel{\widehat{=}} skip \ \textit{iff} \ lab \in \{fence, write(x,r), write(x,n)\} \\
op^{sc}(lab) &\mathrel{\widehat{=}} write_{sc}(x,r) \ \textit{iff} \ lab = flush \wedge v = (\ell, ssb) \\
&\qquad \wedge ssb = \langle (x,r) \rangle \frown tail(ssb) \\
op^{sc}(lab) &\mathrel{\widehat{=}} r_{aux} := r \ \textit{iff} \ lab = (write(x, r_{aux}) \\
&\qquad \wedge reg' = reg[r_{aux} \mapsto reg(r)]) \\
op^{sc}(lab) &\mathrel{\widehat{=}} read_{sc}(x,r) \ \textit{iff} \ lab = read_{mem}(x,r) \\
op^{sc}(lab) &\mathrel{\widehat{=}} r := r_{src} \ \textit{iff} \ lab = read_{loc}(x,r) \wedge v = (\ell, ssb) \\
&\qquad \wedge r_{src} = latest(x, ssb) \\
op^{sc}(lab) &\mathrel{\widehat{=}} lab \qquad \text{else}
\end{aligned}
$$

The transformation of a program into its SC form is then defined as

$$tso2sc(P) \mathrel{\widehat{=}} prog(sg(P))$$

As a preparatory step to this, we might need to bring P into a form without unfenced wd-chains as described in the previous section. For process 1 of Fig. 1 and its store buffer graph in Fig. 3, its SC version is the following:

$$
\begin{aligned}
Init &\mathrel{\widehat{=}} pc = (1, \langle\,\rangle) \\
COp^1 &\mathrel{\widehat{=}} pc = (1, \langle\,\rangle) \wedge pc' = (2, \langle (x,1) \rangle) \wedge \ skip
\end{aligned}
$$

$$COp^2 \mathrel{\widehat{=}} pc = (2, \langle (x,1) \rangle) \wedge pc' = (2, \langle\,\rangle) \wedge \ write_{sc}(x,1)$$
$$COp^3 \mathrel{\widehat{=}} pc = (2, \langle\,\rangle) \wedge pc' = (3, \langle\,\rangle) \wedge \ read_{sc}(y, r1)$$
$$COp^4 \mathrel{\widehat{=}} pc = (2, \langle (x,1) \rangle) \wedge pc' = (3, \langle (x,1) \rangle) \wedge \ read_{sc}(y, r1)$$
$$COp^5 \mathrel{\widehat{=}} pc = (3, \langle (x,1) \rangle) \wedge pc' = (3, \langle\,\rangle) \wedge \ write_{sc}(x,1)$$

In parallel with the SC version of process 2, this can then be given to standard SC verification tools. Our approach is *compositional*: transformations of processes can be done without considering other parallel processes (see Theorem 1); we can reuse transformation results when processes are combined in different ways.

Our main result showing the soundness of this approach is the equivalence of P and $tso2sc(P)$ with respect to local bisimulation.

Theorem 2. *Let P be a program with fenced or write-free loops only and with no unfenced wd-chains. Then*

$$[\![P]\!]_{tso} \approx_\ell [\![tso2sc(P)]\!]_{sc}.$$

Proof sketch: The proof proceeds by defining a relation on the states of $[\![P]\!]_{tso}$ and $[\![tso2sc(P)]\!]_{sc}$. For this, we need some concretisation function for symbolic store buffer contents, concretising the value of a symbolic store buffer ssb with respect to the current state s:

$$conc_s(\langle\,\rangle) = \langle\,\rangle$$
$$conc_s(\langle (x,n) \rangle \frown ssb) = \langle (x,n) \rangle \frown conc_s(ssb) \text{ for } n \in \mathbb{N}$$
$$conc_s(\langle (x,r) \rangle \frown ssb) = \langle (x, s(reg)(r)) \rangle \frown conc_s(ssb) \text{ for } r \in Reg$$

The relation proving bisimilarity is then:

$$\begin{aligned} \mathcal{R} = \{(s,q) \mid\ & first(q(pc)) = s(pc) \\ & \wedge conc_s(second(q(pc))) = s(sb) \\ & \wedge \forall r \in Reg : s(reg)(r) = q(reg)(r)\} \end{aligned}$$

where $first((\ell, ssb)) = \ell$ and $second((\ell, ssb)) = ssb$. □

This result allows us to re-use standard verification techniques for SC programs, might these be automatic or interactive.

6 Experimental Results

In our experiments we wanted to see whether the possibility of using standard SC tools for verification, opened up by our transformation technique, might now have to be paid by an increase in time and space usage of the tools. Our experimental setup was as follows. We used SPIN [13] as model checking tool, both for the SC semantics and for the TSO semantics. SC semantics are provided by SPIN. For TSO, we manually enhanced programs with store buffers and flush and fence operations, mimicking the TSO semantics (see [19] for details). The

experiments all started with a C or C++ program which was compiled to an intermediate representation (IR) with the LLVM compiler. The IR code was then translated to Promela code (input to SPIN) with store buffers. Furthermore, we constructed the sb-graph automatically and out of this the transformed SC program, manually. Implementations of manual steps are on the way. The transformation is linear in the size of the sb-graph and hence, negligible compared to the actual verification effort. The latter depends on a model checker's ability to explore state space or the program complexity in case of a formal proof.

For our experiments we considered a number of mutual exclusion algorithms with two processes each (Dekker, Peterson, Lamport Bakery, Szymanski) and two concurrent data structures, a work-stealing queue by Arora et al. [3] and a stack implementation by Treiber [21]. The latter two allow for different instances in which the processes execute different operations. An instance $UO||TT$ e.g. describes two processes, one doing operations *pushBottom* followed by *popBottom* and the other executing two *popTop* operations. The mutual exclusion algorithms are known to be incorrect under weak memory models and hence, we used both the original unfenced and the correct fenced version. Only one of the examples did not fall into the category of programs with fenced or write-free loops (the unfenced version of Dekker's algorithm). The other examples either just have reading loops, or have implicit fence operations. An implicit fence is for instance generated by a CAS instruction (an atomic compare and swap), which is often used as the only synchronisation primitive in otherwise lock-free data structures. All tests were performed on a virtual machine, Ubuntu Linux, Intel Core i5, 2.53 GHz and 3 GB dedicated to SPIN 6.2.3. All models used for the verification can be found in our repositories at Github[3].

Table 1 provides our verification results giving verification time and number of states generated by SPIN for both the TSO and transformed SC programs. It also gives the number of nodes in the store buffer graph (for the processes viz. operations in the program). The experiments show that our transformation can in a lot of cases actually reduce the state space and verification time. Besides being able to use an SC tool, we can thus furthermore gain time and space when applying the TSO to SC transformation. Compared to the work of [5], who also used a transformation technique and in their experiments looked at these mutual exclusion algorithms, we can moreover state that the runtime of our approach is significantly smaller. The results are, however, not directly comparable since they used different verification tools.

In our previous work [20], we proved linearizability [12] of the Burns mutual exclusion algorithm [8] under TSO using an interactive theorem prover. Particularly, we compared the proof effort of (1) a program encoding TSO with explicit store buffers against (2) a transformed program version using SC semantics (based on the idea that we formalized in this paper). For the Burns mutex, the transformation reduced complexity (invariant size) and proof effort (number of proof steps) approximately by half. Our transformation technique can thus also be helpful in interactive proving.

[3] https://github.com/oleg82upb.

Table 1. Verification results for case studies.

Algorithm (each for 2 processes)	TSO Model		Transformed SC Program		
	states	time [s]	states	time [s]	nodes#
Dekker (fenced)	655	≈ 0	147	≈ 0	17
Dekker (unfenced)	540	≈ 0	unfenced writing loop		
Peterson (fenced)	709	≈ 0	270	≈ 0	14
Peterson (unfenced)	805	≈ 0	1,271	0.01	28
Lamport Bakery (fenced)	2,907	0.01	405	0.01	24
Lamport Bakery (unfenced)	16,087	0.17	163	≈ 0	75
Szymanski (fenced)	2,778	0.01	1,741	≈ 0	30/32
Szymanski (unfenced)	923	0.01	171	≈ 0	55
Work-Stealing Queue (fenced) UOUOUOU‖TTT $\widehat{=}$ pushBottom, O $\widehat{=}$ popBottom, T $\widehat{=}$ popTop (stealing process)	73,703	0.32	86,566	0.23	13/46/18
Treiber-Stack UUUOOO ‖ OOOUUU U $\widehat{=}$ push, O $\widehat{=}$ pop	1,913,313	8.93	1,821,426	4.68	29/15

7 Related Work

In the last years, several approaches were proposed in order to deal with software verification under the influence of weak memory models, ranging from theoretical results to practical techniques.

Atig et al. [4] have shown that the reachability problem for programs in a TSO or PSO environment is decidable via reduction to lossy channel machines. However, for other relaxed memory models like RMO, the problem is undecidable. Bouajjani et al. [7] determined the complexity (PSpace) of deciding robustness of programs against TSO. Two recent approaches [1,6] provide underapproximating techniques for checking program correctness under TSO.

Several approaches [2,5,9,15] propose reduction techniques, which allow for a reuse of verification techniques developed for SC. The approach closest to us is the one by Atig et al. [5]. Similar to us, they provide a translation from a TSO program to an equivalent SC program, but assuming an age bound k. The bound k stems from the observation that store buffer entries can stay for at most k steps in the store buffer until they are eventually flushed to the memory. Their approach is, thus, to model the store buffer behavior as part of the new SC program by introducing k vectors of shared variable copies as part of the local state. Hence, rather than getting rid of the complexity of store buffers, store buffers are replaced with auxiliary vectors in the SC program. The bound results in some sort of bounded verification; if the program exceeds the bound (e.g., in case of loops without fences), the bound needs to be increased and verification restarted.

In our approach (auxiliary) variable copies are only used if they are indeed required, i.e., when the symbolic store buffer entry of a write source of a wd-chain can be redefined between write and flush. We have at most one new variable per

register in the program. This is enough since we consider a restricted class of programs (fenced loops only) for which we can then carry out a (non-bounded) verification. In summary, our approach works for a restricted class of programs, but for this carries out a full verification, whereas Atig et al.'s technique works for all programs, however, sometimes only with an underapproximating analysis. For the class of programs with fenced-loops our approach furthermore generates fewer auxiliary variables, and – as the experiments show – may speed up verification. We thus see our approach as an excellent alternative to Atig et al.'s, in case the program falls into our category of fenced-loop programs.

8 Conclusion

In this paper, we have presented a simple and practical reduction of program verification under the influence of TSO to an SC setting via program transformation. Consequently, the transformed program can be verified using common techniques assuming a sequential consistent memory model.

Our transformation exploits that most of the non-determinism inherent to TSO can be computed statically, which is captured by a store buffer graph in our approach. By encoding the store buffer graph into an equivalent SC program, we completely get rid of store buffers and the burden of reasoning about them. Our experiments show that the transformation can even simplify verification (using a model checker and a theorem prover) of programs under TSO.

Our approach is restricted to programs with at least one fence in loops containing writes. The reason to restrict ourselves to this class of programs is that they can be represented by a finite store buffer graph. We could extend our approach to also deal with unfenced writing loops in a setting of bounded store buffers. However, we would then have to introduce multiple copies of register variables in the new program corresponding to different loop iterations in the original program. In principle, we would then arrive at a technique similar to [5].

References

1. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015)
2. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 512–532. Springer, Heidelberg (2013)
3. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. Theor. Comput. Syst. **34**(2), 115–144 (2001)
4. 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 2010, pp. 7–18. ACM (2010)
5. Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in TSO analysis. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 99–115. Springer, Heidelberg (2011)

6. Bouajjani, A., Calin, G., Derevenetc, E., Meyer, R.: Lazy TSO reachability. In: Egyed, A., Schaefer, I. (eds.) FASE 2015. LNCS, vol. 9033, pp. 267–282. Springer, Heidelberg (2015)
7. Bouajjani, A., Meyer, R., Möhlmann, E.: Deciding robustness against total store ordering. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 428–440. Springer, Heidelberg (2011)
8. Burns, J., Lynch, N.A.: Mutual exclusion using indivisible reads and writes. In: 18th Allerton Conference on Communication, Control, and Computing, pp. 833–842 (1980)
9. Cohen, E., Schirmer, B.: From total store order to sequential consistency: a practical reduction theorem. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 403–418. Springer, Heidelberg (2010)
10. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4), 451–490 (1991)
11. Dan, A.M., Meshman, Y., Vechev, M., Yahav, E.: Predicate abstraction for relaxed memory models. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis. LNCS, vol. 7935, pp. 84–104. Springer, Heidelberg (2013)
12. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
13. Holzmann, G.J.: The SPIN Model Checker - Primer and Reference Manual. Addison-Wesley, Boston (2004)
14. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9), 690–691 (1979)
15. Linden, A., Wolper, P.: An automata-based symbolic approach for verifying programs on relaxed memory models. In: van de Pol, J., Weber, M. (eds.) Model Checking Software. LNCS, vol. 6349, pp. 212–226. Springer, Heidelberg (2010)
16. Milner, R. (ed.): A Calculus of Communicating Systems. Springer, Heidelberg (1980)
17. Pasareanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. STTT **11**(4), 339–353 (2009)
18. 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**(7), 89–97 (2010)
19. Travkin, O., Mütze, A., Wehrheim, H.: SPIN as a linearizability checker under weak memory models. In: Bertacco, V., Legay, A. (eds.) HVC 2013. LNCS, vol. 8244, pp. 311–326. Springer, Heidelberg (2013)
20. Travkin, O., Wehrheim, H.: Handling TSO in mechanized linearizability proofs. In: Yahav, E. (ed.) HVC 2014. LNCS, vol. 8855, pp. 132–147. Springer, Heidelberg (2014)
21. Treiber, R.K.: Systems programming: coping with parallelism. Technical report RJ 5118, IBM Almaden Res. Ctr. (1986)
22. Wonisch, D., Schremmer, A., Wehrheim, H.: Programs from proofs – a PCC alternative. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 912–927. Springer, Heidelberg (2013)
23. Yang, Y., Gopalakrishnan, G., Lindstrom, G.: UMM: an operational memory model specification framework with integrated model checking capability. Concurrency Comput. Pract. Experience **17**(5–6), 465–487 (2005)