Handling TSO in Mechanized Linearizability Proofs

Oleg Travkin and Heike Wehrheim

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

Abstract. Linearizability is the key correctness criterion for concurrent data structures. In recent years, numerous verification techniques for linearizability have been developed, ranging from model checking to mechanized proving. Today, these verification techniques are challenged by the fact that concurrent software is most likely to be run on multi-core processors equipped with a weak memory semantics (like total store order, TSO), making standard techniques unsound. While for model checking and static analysis techniques, approaches for handling weak memory in verification have already emerged, this is lacking for theorem-prover supported, mechanized correctness proofs.

In this paper, we present the very first approaches to handling TSO semantics in mechanized proofs of linearizability. More precisely, we introduce two approaches, one explicitly modelling store buffers and a second avoiding this modelling by instead reordering program operations. We exemplify and discuss our approach on two case studies, Burns mutual exclusion algorithm and a work stealing dequeue of Arora et al., both of which require additional memory barriers when executed on TSO.

Keywords: Linearizability, weak memory models, verification, TSO, KIV.

1 Introduction

With the advent of multi-core processors and the consequently rising increase in concurrent software, high performance concurrent data structures have come into the focus of algorithm designers. Concurrent data structures allow for a concurrent access to standard data structures like lists, queues or stacks. High performance is achieved by (mostly) avoiding locks, and instead relying on very fine-grained atomicity. Due to the subtlety of lock-free algorithms, their proof of correctness can be exceptionally complex. The quasi-standard correctness criterion for concurrent data structures is *linearizability* [18]. Many techniques for the verification of linearizability emerged in the past, ranging from manual proofs (usually done by the algorithm designers themselves), to model checking [29] and theorem proving [25,28].

A large number of existing verification techniques, both for concurrent software in general and more specifically for linearizability, assume a sequentially consistent memory model (SC) [21], i.e., assume statements in a sequential program to be executed in program order and concurrent programs to be an interleaving of components. However, multicore processors like x86, SPARC or POWER provide weaker execution semantics than SC and allow executions to deviate from program order [2]. The reason behind these out-of-order execution is (mainly) the use of *store buffers* attached to processor cores. Store buffers can delay write instructions while later instructions w.r.t. program order are further processed. As a consequence, an execution may appear as though out-of-order. Verification techniques coping with weak memory can so far be classified into two strands. The first strand aims at reusing verification technique for sequential consistency. This starts with techniques for detecting non-sequentially consistent behaviour (monitoring, testing, robustness checking [10,11,8]) which can then be eliminated by fence insertion (e.g. using techniques of [1,20]), or finding program structures which guarantee SC behaviour even for relaxed memory models (like data race freedom or triangular race freedom [22]). The other strand of research takes weak memory behaviour into account, either by explicitly modelling store buffers [9] or by rewriting the program in such a way that an SC-based verification becomes sound [6,3,14]. None of these approaches have, however, proposed techniques for handling weak memory within mechanized proofs of linearizability. The advantage of a mechanized proof is the establishment of correctness for *arbitrary uses* of the data structure, i.e. arbitrary method invocations by an *arbitrary number* of processes. Compared to model checking or testing approaches, mechanized proofs are not limited to specific usage scenarios.

In this paper, we propose two approaches for handling weak memory semantics in mechanized correctness proofs of linearizability. The first approach builds on an explicit modelling of store buffers and delayed writes. Unlike model checking approaches, we need not (but could) assume bounds on the buffer size in our models. The second approach employs an explicit reordering of program statements as to mimic the behaviour of store buffers. It turns out that the second approach is more convenient for mechanized proving as it keeps us from having to define and reason about invariants on the store buffer contents. In our definition of linearizability we follow [17,7] in that we compare the implementations of concurrent data structures run on TSO against sequential specifications interpreted in an SC way (TSO-to-SC linearizability). Our general proof principle proceeds by showing a *simulation* relation to exist between implementation and specification, and follows established simulation-based proof techniques for linearizability on SC [15,25].

We discuss and exemplify our approach(es) on two case studies, a variant of Burns mutual exclusion algorithm [12] and a work-stealing double-ended queue of Arora et al. [5]. While the first example is rather small and mainly used for demonstration purposes, the second example realistically reflects the size of modern concurrent data structures. Both examples are non-linearizable when executed on a TSO architecture in their original form and need additional memory barriers for soundness. We were able to prove linearizability of the fenced versions for both examples, using the theorem prover KIV [24].

2 Background

TSO Architecture. Nowadays, one of the most wide-spread multicore processor architecture is the x86 [19,4], which provides the TSO memory model. Figure 1 illustrates the architecture of a modern multicore processor providing a TSO memory model. Each processor has a write buffer to store its writes before they are (later) flushed to shared memory. Reading of variables either takes place from the buffer (if there is a pending write of this variable in the store buffer) or from shared memory. Memory barriers (or *fence* operations) can be used to block program execution until the store buffer is completely flushed.

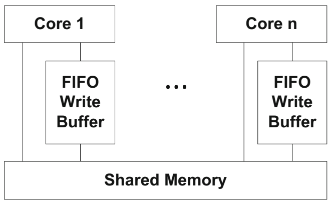

Fig. 1. TSO architecture as common for x86-based multicore processors.

$\mathit{Initially} : x = 0 \wedge y = 0$

Process 1	Process 2
$\mathit{write}(x, 1);$	$\mathit{write}(y, 1);$
$\mathit{read}(y, r1);$	$\mathit{read}(x, r2);$

$r1 = 0 \wedge r2 = 0$

Fig. 2. Test program for detection of *Write* $\rightarrow$ *Read* reordering, also known as litmus test

As a consequence of this architecture, TSO exhibits two relaxations of program order compared to SC. First, writes may appear as if they were executed after a later read, i.e., the order *Write* $\rightarrow$ *Read* is relaxed. This can happen when write and read access different memory locations. Figure 2 shows a test program for detection of this behavior. Initially both shared variables x and y hold the value 0. The test detects reordering if both registers have values $r1 = 0 \wedge r2 = 0$ at the end of its execution and hence at least one process must have had its instructions reordered. Simple interleaving, as in an SC setting, does not allow this outcome. A second relaxation allows processes to read their own writes early. If a write buffer contains a pending write to an address requested by a read, the value from the buffer is read. This behavior is called *early-read* [2] or *Intra-Process-Forwarding* [19], because a reading processor is allowed to see its own writes before they are committed to the memory and hence before other processes can see them.

Burns Algorithm. Our objective is to show (a certain form of) correctness of algorithms executed on weak memory models. The correctness proofs thus will need to take the unusual non-SC semantics into account. We will exemplify our approach on the following mutual exclusion algorithm of [12]. Originally, the algorithm was defined with a loop for each process, in which it tries to enter and leave a critical section. We modified the algorithm slightly (as to be able

```
bool *flag0 = 0, *flag1 = 0;

//process0:                          //process1:
void acquire0{                       void acquire1{
    *flag0 = 1;                 retry: while (*flag0 != 0) {
    /*need fence here*/                    /*wait*/
    while (*flag1 != 0) {              }
        /*wait*/                       *flag1 = 1;
    }                                  /*need fence here*/
}                                      if(*flag0 != 0) {
                                           *flag1 = 0;
void release0() {                          goto retry;
    *flag0 = 0;                        }
}                                    }

                                     void release1() {
                                         *flag1 = 0;
                                     }
```

Fig. 3. Mutual exclusion algorithm for two processes (based on [12])

to view it as a concurrent data structure) by explicitly defining two operations — *acquire* and *release* — which can then be repeatedly called (in turn). The Burns algorithm (see Fig. 3) uses a flag for each process to indicate its intention of a process to enter the critical section. Both flags are initially 0 and set to 1, when a process tries to enter. It is an asymmetric algorithm in the sense that processes are ordered in terms of priority. Process $p0$ (highest priority) sets its flag to $\mathit{flag}0 = 1$ and waits until it observes $\mathit{flag}1 = 0$. In this case, process $p1$ is not trying to enter the critical section and will not enter until $p0$ has left it. In contrast to $p0$, $p1$ checks the flag of $p0$ before setting its own flag and checks $\mathit{flag}0$ again after having set $\mathit{flag}1$ to 1. If $\mathit{flag}0$ changes in the meantime, $p1$ resets its flag to 0 (allowing $p0$ to enter and finish) and retries. Otherwise, $p1$ finishes by entering the critical section. Both processes release their ownership by setting their flag to 0.

In order to determine possible effects of weak memory on the execution of this algorithm, we first of all need to explicitly see the low-level reads and writes. To this end, we first compile a C program into an intermediate representation, here using the LLVM[1] compiler framework with intermediate representation LLVM IR. On this, we can see the atomic reads and writes. Figure 4 shows the compiled code for the operation acquire0. The code defines a function which is structured into labeled blocks (entry, cond, body and end). Global variables (here, the two flags) are prefixed with @. Local registers are prefixed with %. The local variables %tobool, %conv and %cmp are just used to store values of type conversions (the first two) and the value of a comparison. We thus will not explicitly model these later. The br instruction is either a simple jump (e.g., in block entry) or a conditional jump (e.g., in block cond with variable %cmp being the boolean condition). Instruction load (resp. store) corresponds to a read (resp. write) of a global variable. When determining the semantics of this LLVM-IR code on TSO, we thus need to assign these statements a non-standard semantics.

[1] www.llvm.org

```
define void @_Z5acquire0() nounwind {
entry:
  store i8 1, i8* @flag0
; ---- need fence here ----
  br label %cond

cond:
  %0 = load i8* @flag1
  %tobool = trunc i8 %0 to i1
  %conv = zext i1 %tobool to i32
  %cmp = icmp ne i32 %conv, 0
  br i1 %cmp, label %cond, label %end

end:
  ret void
}
```

Fig. 4. LLVM IR code for method acquire0 after compilation

$$\begin{aligned}
COP_1 &\mathrel{\widehat{=}} ls.pc = A1_0 \ \wedge\ ls'.pc = A2_0\\
&\qquad \wedge\ write((flag0, 1), ls, mem, ls', mem')\\
COP_2 &\mathrel{\widehat{=}} ls.pc = A2_0 \ \wedge\ ls'.pc = A3_0\\
&\qquad \wedge\ fence(ls, mem, ls', mem')\\
COP_3 &\mathrel{\widehat{=}} ls.pc = A3_0 \ \wedge\ ls'.pc = A4_0\\
&\qquad \wedge\ read((flag1, f1), ls, mem, ls', mem')\\
COP_{4a} &\mathrel{\widehat{=}} ls.pc = A4_0 \ \wedge\ ls.f1 \neq 0 \ \wedge\ ls'.pc = A3_0\\
COP_{4b} &\mathrel{\widehat{=}} ls.pc = A4_0 \ \wedge\ ls.f1 = 0 \ \wedge\ ls'.pc = A5_0\\
COP_{flush} &\mathrel{\widehat{=}} flush(ls, mem, ls', mem')
\end{aligned}$$

Fig. 5. Encoding of program behavior for method acquire0. Parameters (mem, ls, mem', ls') of each COP predicate were omitted for brevity.

3 TSO model

We are ultimately interested in a mechanized proof of correctness of algorithms. To this end, we first need a precise formal model of TSO on top of which we can then define the semantics of programs. We model shared memory as a function $mem : \mathbb{N} \to (\mathbb{N} \cup null)$, where we use $\mathbb{N}$ as the memory address space and allow $\mathbb{N}$ or $null$ to be the result of a memory access. The following three axioms define memory access (written as $mem[n]$) and modification ($mem[n, a]$ modifying memory mem at address n to become a):

$$\vdash\ mem = mem0 \ \Leftrightarrow\ \forall\, n \bullet mem[n] = mem0[n] \tag{1}$$

$$\vdash\ mem[n, a][n] = a \tag{2}$$

$$\vdash\ n \neq n0 \ \Rightarrow\ mem[n0, a][n] = mem[n] \tag{3}$$

(1) defines the identity of two memory functions, (2) states that access to the address n will yield the last value written to it, and (3) states that modifying one address will not change the value of another address.

In order to fix the semantics of programs on TSO, we first define an instruction set for the interaction with the memory and store buffer (similar to [26]). Instructions affecting store buffer and memory are *write*, *read* and *fence* explicitly appearing as operations in programs plus *flush* which is occasionally executed as to flush the store buffer. We let P be the set of all process identifiers, and write ls to describe the *local state* of a process $p \in P$. The local state comprises the process identifier $ls.p \in P$, the store buffer $ls.buf \in (\mathbb{N} \times (\mathbb{N} \cup null))^*$, values of local registers $ls.r$ from some set of registers Reg and a program counter $ls.pc$ from some set PC. We use LS to denote the set of local states. The instructions modify the state as follows (where $+$ is concatenation and primed variables are used to describe the after state):

$$\begin{aligned} \mathit{write}((n,a), \mathit{ls}, \mathit{mem}, \mathit{ls}', \mathit{mem}') \Leftrightarrow\ & \mathit{ls}'.\mathit{buf} = \mathit{ls}.\mathit{buf} + (n,a) \\ & \wedge\ \mathit{mem}' = \mathit{mem} \end{aligned} \tag{4}$$

$$\begin{aligned} \mathit{flush}(\mathit{ls}, \mathit{mem}, \mathit{ls}', \mathit{mem}') \Leftrightarrow\ & \mathit{ls}.\mathit{buf} = (n,a) + \mathit{ls}'.\mathit{buf} \\ & \wedge\ \mathit{mem}' = \mathit{mem}[n,a] \end{aligned} \tag{5}$$

For the definition of a read, we need a helper function $\mathit{latest}(n, \mathit{buf})$ (not given here) to determine the latest entry for the requested address in the buffer.

$$\begin{aligned} \mathit{read}((n,r), \mathit{ls}, \mathit{mem}, \mathit{ls}', \mathit{mem}') \ \Leftrightarrow\ & \mathit{ls}'.\mathit{buf} = \mathit{ls}.\mathit{buf}\ \wedge\ \mathit{mem}' = \mathit{mem}\ \wedge \\ & \textbf{if } n \in \mathit{ls}.\mathit{buf} \textbf{ then } \mathit{ls}'.r = \mathit{latest}(n, \mathit{ls}.\mathit{buf}) \\ & \textbf{else } \mathit{ls}'.r = \mathit{mem}[n] \end{aligned} \tag{6}$$

A read either obtains the latest value from the store buffer, if there is one, or it obtains the value directly from the memory. The buffer and memory remain unmodified. Finally, fences in the program code block program execution until the store buffer is emptied. To this end, the fence is only enabled when $\mathit{ls}.\mathit{buf} = \langle\,\rangle$ and blocks execution otherwise.

$$\mathit{fence}(\mathit{ls}, \mathit{mem}, \mathit{ls}', \mathit{mem}') \Leftrightarrow \mathit{ls}.\mathit{buf} = \langle\,\rangle \wedge \mathit{ls}'.\mathit{buf} = \mathit{ls}.\mathit{buf} \wedge \mathit{mem}' = \mathit{mem} \tag{7}$$

It is in the semantics of these instructions (and thus of the load and store in LLVM) where the difference to SC semantics can be found. For modelling the behavior of a given program we next proceed as follows. For the Burns algorithm, we fix the set Reg of registers and assign the register %0 used in acquire0 a name (here, $f1$ because it stores the value of $\mathit{flag}1$). For the memory, we use the global variable names $\mathit{flag}0$ and $\mathit{flag}1$ as constants 0 and 1 to access mem.

Figure 5 shows the encoding for the method acquire0 in Figure 4. In principle, we define one operation per program instruction. However, as LLVM-IR contains a lot of operations which need not be modelled in the theorem prover we use (e.g., type conversions), we get more compact operations in our model. All operations are modelled as predicates. By specifying the change of the program counter, we define the control flow of the method. After invocation, the program counter is at $A1_0$. The first instruction (store) changes the program counter to $A2_0$ and attempts to write value 1 into address flag1. Note that the write does not modify the memory directly, but enqueues the address value pair to the store buffer. We ignore the following br instruction since it is just a jump to next instruction in program order. Operation COP_2 is a fence instruction which we will need further on, but which first of all is not part of the operations. Please note that local instructions, e.g., the four instructions after the load in Figure 4, can be composed to a single one, because they are invisible to other processes and hence, their atomicity is irrelevant for the correctness of the algorithm. The predicate $\mathit{COP}_{\mathit{flush}}$ models the non-deterministic flushes of the store buffer. It is not restricted to any particular program location and can be performed repeatedly.

4 Proving Linearizability

Our main interest is in proving linearizability of concurrent data structures. Linearizability is a correctness condition for concurrent data structures which states that — when used concurrently — the data structure acts as though used sequentially. To prove this, we need to find an "equivalent" sequential execution for every concurrent run. An execution, or *history*, consists of a sequence of invocations and responses of methods, e.g. of the acquire and release of Burns algorithm. Every concurrent history, i.e., history in which more than one method might run at a time, has to have a matching sequential history preserving the order of operations from the concurrent history. For a formal definition see [18]. Linearizability is often explained in terms of *linearization points* (LPs) which are points within methods where the real effect of the methods seems to take place atomically. The acquire methods of Burns algorithm pass their linearization point when they observe the flag of the other process to be zero. The release methods have their LPs, when the write to their flag becomes visible.

There are a number of different ways of formally proving linearizability for a given data structure. Here, we intend to prove linearizability by showing that the algorithm's implementation *simulates* a sequential specification of the data structure (following approaches in [15,25]). In the sequential specification all operations are executed atomically, and thus the sequential specification only has sequential histories. The proof needs to build up a *simulation relation* between our behavior model of the algorithm and another sequential model. We also call this the *concrete* and the *abstract* model. The concrete model has concrete operations (called $COP_{\dots}$), which we have already seen, and the abstract model has abstract operations. For the Burns algorithm, we have an abstract state space simply consisting of one variable $mtx \in (\{none\} \cup P)$ and operations acquire and release for each process.

$$\begin{aligned}
AOP_{acquire0} &\mathrel{\widehat{=}} (mtx = none \wedge mtx' = 0)\\
AOP_{acquire1} &\mathrel{\widehat{=}} (mtx = none \wedge mtx' = 1)\\
AOP_{release0} &\mathrel{\widehat{=}} (mtx = 0 \wedge mtx' = none)\\
AOP_{release1} &\mathrel{\widehat{=}} (mtx = 1 \wedge mtx' = none)
\end{aligned}$$

Thus, in the abstract model, we have atomic operations corresponding to methods, and, in the concrete model, these are implemented by lots of concrete operations, some of which are LPs. Formally, we thus have a non-atomic refinement between abstract and concrete model which we intend to prove via a forward simulation. For showing the existence of a forward simulation, we first need to define an abstraction relation Abs between the state space of the abstract model (here, variable mtx) and that of the concrete model (here, global variable mem plus all local states ls of processes). In our case, the abstraction relation will be a *function* from concrete to abstract. Second, we need to define the linearization points of methods[2]. All concrete steps COP which are not LPs have to be shown

[2] In general, simulation proofs can also be done when LPs are not fixed, but for the algorithms in this paper this is not necessary.

to simulate abstract skip steps (empty operations) while the LP steps have to simulate the corresponding abstract operation. In case of the method acquire0, the LP is at COP_3, but only if the method observes $\mathit{flag}1 = 0$. Hence, COP_3 observing $\mathit{flag}1 = 0$ has to simulate $AOP_{acquire0}(mtx, mtx')$. All other acquire0 operations have to simulate skip steps. Our proof technique then proceeds by locally reasoning about processes.

The main idea behind the local proof obligations (LPO) (see [15]) is to prove linearizability for two processes, where one process p is explicit and the other process q a symbolic representation of all other processes. Both processes operate on the shared global state $gs \in GS$, which in our case studies is the memory function *mem*. The local states of both processes p and q are $lsp, lsq \in LS$. In addition, we need to define and establish an *invariant INV* on global and local states. The following is one of a number of proof obligations which need to be shown for simulation.

$$\begin{array}{l}\forall\, gs, gs' : GS, lsp, lsq, lsp' : LS \bullet \\ \quad INV(gs, lsp) \wedge INV(gs, lsq) \wedge COP(gs, lsp, gs', lsp') \\ \quad \Rightarrow \\ \quad INV(gs', lsp') \wedge INV(gs', lsq) \wedge AOP_{pq}(Abs(gs, lsp, lsq), Abs(gs', lsp', lsq))\end{array}$$

This proof obligation states the following condition: if the invariant holds both for process p and the other process q, and p executes operation COP thereby changing the global state and its local state, then the invariant still holds for p and q afterwards and a corresponding abstract operation can be executed on the corresponding abstract states[3]. If a particular COP-transition is a linearization point (LP), then AOP_{pq} must be the corresponding abstract operation, and a skip step, otherwise. Depending on which process passes its linearization point, AOP_{pq} can be an abstract operation performed by either process p or q or both. The latter two cases can occur by process p helping other processes to finish their operation or by p passing its own linearization point and by doing this causing the other process to linearize as well.

Next, we apply this technique to our running example. However, the first observation (found by using the model checking approach [27]) is that the acquire methods of the Burns algorithm both need a fence (see Fig. 3). Otherwise, the initial write could be still pending while the flag of the other process is read within the loop. Hence, both processes would be able to enter the critical section at the same time by observing the other flag value to be zero while the write to the own flag is still pending. In particular, the following history of invokes and returns would be possible:

$$\langle inv_0(acquire0), inv_1(acquire1), ret_0(acquire0), ret_1(acquire1)\rangle$$

which corresponds to one of the sequences:

$$AOP_{acquire0};\ AOP_{acquire1} \text{ or } AOP_{acquire1};\ AOP_{acquire0}$$

[3] In principle, Abs is only defined on gs. If information about local states is needed for the definition of Abs, these have to moved into the global state via auxiliary variables.

Both sequences violate the corresponding AOP definitions, because both require $mtx = none$, but modify its value. Hence, the second AOP must not finish until a release method linearizes. We place a fence at COP_2 in acquire0 in order to ensure the write is no longer pending during observation of the other flag. Thereby, we disable executions as the one mentioned above. We modify acquire1 similarly.

Now that we fixed the implementation by ruling out non-linearizable executions, we can define the invariant, which is defined as properties holding at a particular program location. In case of the Burns algorithm, we are interested in the values of the flags. However, the flag values depend on the state of the store buffer, i.e., whether a write to the flag was flushed or not. Thus, we have to specify two kinds of properties in our invariant: First, the invariant has to establish the possible states of the store buffer at particular program locations. Second, the possible flag values depending on the store buffer state and the program have to be specified. For the method acquire0 the invariant is defined as:

$$\begin{aligned} INV(mem, ls) \widehat{=} & ((ls.pc \in \{A1_0, A3_0, A4_0, A5_0\} \Rightarrow ls.buf = \langle\,\rangle) \\ & \wedge (ls.pc = A2_0 \Rightarrow ls.buf = \langle\,\rangle \vee ls.buf = \langle(flag0, 1)\rangle) \\ & \wedge (ls.pc = A1_0 \Rightarrow mem[flag0] = 0) \\ & \wedge (ls.pc = A2_0 \wedge ls.buf = \langle\,\rangle \Rightarrow mem[flag0] = 1) \\ & \wedge (ls.pc = A2_0 \wedge ls.buf = \langle(flag0, 1)\rangle \Rightarrow mem[flag0] = 0) \\ & \wedge ((ls.pc \in \{A3_0, A4_0, A5_0\} \Rightarrow mem[flag0] = 1) \\ & \wedge (ls.pc = A5_0 \Rightarrow ls.f1 = 0) \end{aligned}$$

where program location $A2_0$ is the one with a potentially pending write to $flag0$ and thus having two possible states of the store buffer, which determine the value of $mem[flag0]$. Note that the value of $mem[flag0]$ at the other program locations $(A1_0, A3_0, A4_0, A5_0)$ can only be stated without referring to the store buffer state, because we know that the store buffer is empty. Otherwise, a similar distinction to the one at location $A2_0$ would be necessary.

Finally, we provide an abstraction function Abs that maps each concrete state to an abstract state. Throughout all executions, $flag0 = 1$ (resp. $flag1 = 1$) means that process 0 (resp. 1) is either the owner of the mutex or it tries to acquire it. We distinguish the two cases by taking the progress of local states into account. We use the two range predicates $observed_0(mem[flag1] = 0)$ in order to define the range after process 0 observed $flag1 = 0$ and $observed_1(mem[flag0] = 0)$ for process 1, respectively. The abstraction function is then defined as a case distinction over the three cases:

$$\begin{aligned} Abs(lsp, lsq, mem) \widehat{=} & \textbf{if } mem[flag0] = 1 \wedge observed_0(mem[flag1] = 0) \\ & \quad \textbf{then } mtx = 0 \\ & \textbf{else if } mem[flag1] = 1 \wedge observed_1(mem[flag0] = 0) \\ & \quad \textbf{then } mtx = 1 \\ & \textbf{else } mtx = none \end{aligned}$$

Given the above abstraction function and invariant, we were able to show all proof obligations for the fenced Burns algorithm, thereby establishing linearizability with respect to the given sequential specification.

5 Avoiding Store Buffers

In the last section, we have seen how to prove linearizability using an explicit modelling of store buffers to encode the TSO behavior. However, keeping store buffers as part of the state has a huge drawback. Mechanized proofs reveal many impossible cases of executions which thus need to be ruled out by the invariant (unless they are harmless). Hence, the invariant not only has to cover the properties of potential store buffer states, but also the interconnection between store buffer states and values of global and local variables. Hence, the simplicity of specification due to an operational memory model is paid by the complexity of invariants, which have a major impact on the size of correctness proofs and the time and effort that is required for the proofs.

In the following, we will therefore present an idea of how to transform our program model under TSO into an equivalent program model under SC, for which store buffers are no longer required. For the proof, we use the proof obligations from the previous section in combination with the new program model. First of all, we have to make some restrictions to the class of the programs to which our transformation applies. We restrict our transformation to programs, which are (1) in SSA-form [13], (2) do not read early (from store buffer) and (3) loops must be either non-writing or contain at least one synchronizing instruction (fence, CAS instruction, etc.) that limits the potential reordering to a finite delay. Although the three conditions seem to be a strong limitation to the applicability of our approach, they hold surprisingly often to the best of our experience: (1) is a typical intermediate representation by compilers as in case of the LLVM compiler framework, (2) is rarely relevant for concurrent algorithms that are adapted for weak memory models, because reads to previously written shared variables do usually have synchronization in between in order to ensure that the write is flushed before the read is issued. Condition (3) is the actual limitation of the class of programs, since not all loops will have memory barriers. However, our transformation still applies to a large class of algorithms, since most concurrent algorithms rely on some sort of synchronization primitives.

The transformation proceeds in two steps. The starting point of the transformation are the concrete operations COP_i of some method implementation. These are used to build a *symbolic reachability graph* in the first step. In this graph, the nodes are pairs of program location and symbolic store buffer contents. In a later step, we use this graph as basis for the construction of an equivalent program with its operations having SC semantics, and thus, without the need of store buffers.

Definition 1. *A* symbolic reachability graph $G = (N, E)$ *consists of a set of nodes* $N \subseteq PC \times (\mathbb{N} \times (\mathbb{N} \cup Reg))^*$ *and edges* $E \subseteq N \times Lab \times N$*. The labels of the edges are memory instructions or are empty.*

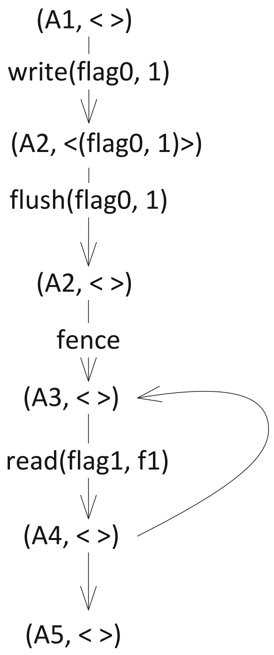

Fig. 6. Abstract reachability graph of operation acquire0

$$\begin{aligned}
COP_{1asc} &\mathrel{\widehat{=}} ls.pc = (A1_0, \langle\,\rangle) \\
&\quad \wedge ls'.pc = (A1_0, \langle(\mathit{flag}0, 1)\rangle) \\
&\quad \wedge mem' = mem \\
COP_{1bsc} &\mathrel{\widehat{=}} ls.pc = A1_0, \langle(\mathit{flag}0, 1)\rangle) \\
&\quad \wedge ls'.pc = (A2_0, \langle\,\rangle) \\
&\quad \wedge mem' = mem[\mathit{flag}0, 1] \\
COP_{2sc} &\mathrel{\widehat{=}} ls.pc = (A2_0, \langle\,\rangle) \\
&\quad \wedge ls'.pc = (A3_0, \langle\,\rangle) \\
&\quad \wedge mem' = mem \\
COP_{3sc} &\mathrel{\widehat{=}} ls.pc = (A3_0, \langle\,\rangle) \\
&\quad \wedge ls'.pc = (A4_0, \langle\,\rangle) \\
&\quad \wedge ls'.f1 = mem[\mathit{flag}1] \\
COP_{4asc} &\mathrel{\widehat{=}} ls.pc = (A4_0, \langle\,\rangle) \\
&\quad \wedge ls.f1 = 0 \wedge ls'.pc = (A5_0, \langle\,\rangle) \\
&\quad \wedge mem' = mem \\
COP_{4bsc} &\mathrel{\widehat{=}} ls.pc = (A4_0, \langle\,\rangle) \\
&\quad \wedge ls.f1 \neq 0 \wedge ls'.pc = (A3_0, \langle\,\rangle) \\
&\quad \wedge mem' = mem
\end{aligned}$$

Fig. 7. Encoding of program behavior for operation *acquire*0

The symbolic store buffer contents either contain pairs of memory address and register name or memory address and constant. The graph of an operation implementation is incrementally constructed as follows. The initial node consists of the initial program location and an empty store buffer. New nodes and edges are constructed as follows:

$(l, buf) \xrightarrow{lab} (l', buf')$ iff $\exists\, COP_i$ such that

- $COP_i = (ls.pc = l \wedge ls'.pc = l' \wedge write((n, r), ls, mem, ls', mem')$ and $buf' = buf \frown \langle(n, r)\rangle$, $lab = write(n, r)$, (ditto constants)
- $COP_i = (ls.pc = l \wedge ls'.pc = l' \wedge r_* := r \wedge write((n, r_*), ls, mem, ls', mem')$ and $buf' = buf \frown \langle(n, r_*)\rangle$, $lab = r_* := r \wedge write(n, r)$
- $COP_i = (ls.pc = l \wedge ls'.pc = l' \wedge read((n, r), ls, mem, ls', mem')$ and $buf' = buf$, $lab = read(n, r)$,
- $COP_i = (ls.pc = l \wedge ls'.pc = l' \wedge \mathit{flush}(ls, mem, ls', mem')$ and $\exists(n, r)$ such that $buf = \langle(n, r)\rangle \frown buf'$, $lab = \mathit{flush}$,
- $COP_i = (ls.pc = l \wedge ls'.pc = l' \wedge \mathit{fence}(ls, mem, ls', mem'))$ and $buf = buf' = \langle\,\rangle$, $lab = \mathit{fence}$,
- $COP_i \Rightarrow (ls.pc = l \wedge ls'.pc = l')$, COP_i is no memory instruction and $buf' = buf$, lab empty.

Operation predicates with more complex structure can be treated in a similar way, e.g. by first logically splitting them into one of the forms of above. Thus, what we are tracking here is just the potential contents of the store buffer, and this only symbolically in that we store which register (or constant) the value must come from. This is similar to symbolic execution [23], however, not tracking all variables. The symbolic reachability graph is finite due to the above mentioned restrictions, in particular, because we have no loops with write operations but without fences. Such a graph can be automatically constructed. For operation acquire0 the graph is given in Figure 6.

Note that in loops, the flush of a $write(x, r)$ can be delayed past the redefinition of r corresponding to the next loop iteration (i.e., synchronization between definition and write of r). Thus, the redefinition of register r also modifies the symbolic store buffer content in our reachability graph. To overcome this problem, we replace COPs with such writes $write(x, r)$ in the program by $r_* := r \wedge write(x, r_*)$ with r_* representing the value of r while the write is still pending. Such cases are the only cases in which we need a second variable instance. However, for both of our case studies this was not necessary.

The second step consists of constructing new concrete operations for the SC execution. Basically, the new operations operate on the same global and local variables, however, without $ls.buf$. Instead, we use the nodes in the symbolic reachability graph as new program locations and define one new operation for every edge in the graph according to the following procedure:

1. For edges $(l, buf) \xrightarrow{lab} (l', buf')$, the predicate of the operation has to contain $ls.pc = (l, buf) \wedge ls'.pc = (l', buf')$,
2. If $lab = write(n, r)$, we add a predicate $mem = mem'$.
3. If $lab = r_* := r \wedge write(n, r_*)$, we add a predicate $r_* := r \wedge mem = mem'$.
4. If $lab = read(n, r)$, we add a predicate $ls'.r = mem[n]$.
5. If $lab = flush$ and $buf = \langle (n, r) \rangle ^\frown buf'$, we add predicate $mem' = mem[n, r]$.
6. If $lab = fence$, we add a predicate $mem = mem'$.
7. If the label of the edge is empty, we re-use the part of the old predicate not refering to program locations.

For the symbolic reachability graph of method acquire0 given in Figure 6, we thus get the operations as depicted in Figure 7.

These two transformation steps have to be applied to every method of the algorithm, i.e., to acquire1, release0 and release1 as well. Together, they form our new concrete SC model which then has to be shown to simulate the (same) abstract model. So far, we have just shown correctness of this transformation, i.e. equivalence of old program on TSO to new program on SC, for the concrete algorithms at hand (Burns and the work-stealing deque). A general correctness proof will be one of our next steps.

6 Evaluation

We used the Burns mutual exclusion algorithm [12] as a toy example to play with our transformation idea as described in the previous section and to compare it against a proof based on an operational encoding of the TSO memory model (see Section 3). After getting the first promising results, we decided to tackle a more realistic case study, the work-stealing deque algorithm by Arora et al. [5]. In particular, we were interested in whether we would be able to prove a more realistic size of case study and therefore applied the transformation based approach to it. The algorithm is an array based queue implementation for thread scheduling and requires fences under weak memory models. The queue implementation is based on fine-grained concurrency primitives, e.g., CAS operations. The provided methods require fences in order to prevent elements from being removed twice. Compared to the 20 LOC of the Burns algorithm, the work stealing deque had 58 LOC in our implementation of it. The C/C++ and LLVM IR code for both implementations[4] and the full linearizability proofs[5] are available for download.

We used the theorem prover KIV [24] for the specification and mechanization of our linearizability proofs. KIV provides a library with the proof obligations (including fully mechanized soundness and completeness proofs) for proving linearizability that our work is based on. Furthermore, KIV allows for automation of proofs and provides strong visualization features, e.g., proof trees and specification dependencies, which are crucial for the understanding of why a proof fails. In the following, we provide our key insights about the presented approaches.

Operational vs. Transformed. The operational encoding of the memory model allows for a straightforward translation of the program code to a program model. The simplicity stems from having no need to think about the potential contents of the store buffers during specification. However, as we figured out in our proof of Burns, the store buffer content becomes crucial anyway. A theorem prover reveals all the cases that are *impossible*, but break the property you try to prove. For instance, the store buffer of process 0 could contain pending writes to $flag1$, although process 0 never writes to $flag1$. Such cases have to be ruled out by the invariant. Thus, we specified the possible store buffer contents for each program location in the invariant. Furthermore, we had to specify whether a flag has a particular value or not as properties depending on the state of store buffer. The more states a store buffer can have, the more complex the invariant can get.

Although the transformation of the program model seemed to be more effort in the first place, it actually reduced our proof effort for several reasons. Since, we had to find out about the possible store buffer states anyway, the construction of the abstract reachability graph did not really increase our effort. The presence of the store buffer as part of the state in the operational encoding basically forced us to reason about a FIFO queue in every step, because an invariant has to be established over all steps of the program. We got rid of this burden by removing the store buffer from the local state, although this was paid by gaining more transitions and program locations in the program behavior. However, some of the transitions became empty transitions (e.g., $COP1_{asc}$ in Fig. 7) and were removed. A second beneficial side effect of store

[4] `http://lina-rmm-verification.googlecode.com/svn/trunk/examples`
[5] `http://linearizability.bplaced.de`

buffer removal was a better automation of the proofs. In particular, the Burns proof based on the operational encoding required 3784 proof steps in KIV of which 201 were manual. The proof based on the transformed program model required 1536 steps of which 63 were manual. The generally lower number of proof steps was also due to the significantly smaller invariant in the transformation based proofs (approx. half the size of the former invariant). By removing store buffer properties and the corresponding case distinctions on the flag values, we got simple properties (e.g. $mem[flag0] = 1$) in certain program location ranges ($ls.pc \in \{(A2_0, \langle\,\rangle)\}, (A3_0, \langle\,\rangle), (A4_0, \langle\,\rangle), (A5_0, \langle\,\rangle))$. The difference in time effort was even bigger, but since many specifications could be reused or needed just a bit of adaption from the operational encoding, a comparison would be unfair.

Work Stealing Deque. We verified the work stealing deque by Arora et al. with the transformation based approach only, but experienced similar benefits from the approach. First, we applied the model checking approach [27] to the example in order to find out, where fences had to be placed in the program and to get an idea of how the algorithm works on a low level. Although the specification in KIV took us just a few days, we spent several weeks to find a correct invariant allowing us to prove the algorithm linearizable. The effort was mainly caused due to iterations of adding invariant properties, trying to establish them within the proof, and in case of a failing proof trying to understand why and to adapt the invariant properties again. We assume that a proof based on the operational encoding would have required more effort because of the complexity due to store buffers. The full linearizability proofs for the work stealing deque required 6923 steps of which 1100 were manual.

7 Conclusion

In this paper, we have presented two approaches for the specification of program behavior under TSO and provided first experimental results on their impact to the proof effort. Both approaches focus on the mechanization of proofs in a theorem prover. The operational encoding, a widely used approach, is modular by keeping a memory model separate from the program specification and therefore allows for straightforward program specification. Proofs based on this approach unfold the full behavior during a proof, but require reasoning about store buffer content, which makes the proof tedious and complex.

The basic principle of employing program transformations to allow for SC-based proofs afterwards has also been followed in [6], however, using different transformations. The transformation in [6] uses a bounded number of shared variable copies in order to simulate store buffer behavior. Our transformation makes reasoning about store buffer content obsolete, without adding a burden to reason about store buffer replacements. We were able to show that our approach reduces the proof effort and complexity (in our experiment by half compared to the operational approach) and also enables the reuse of SC-based techniques. The drawback of our transformation is that it is restricted to a particular class of programs (see Sec. 5).

Since the linearizability theory [15] used in our proofs assumes an SC memory model, our proofs do not cover the case of delays (of store buffer flushes) past the return statement of a method. Thus, we implicitly assume fences at invocation and return of methods in order to be sound. We plan to adapt the linearizability theory (similar to [16]) as to be able to drop this assumption.

Currently, we are working on proving correctness of the program transformation, i.e. proving that the TSO model of the original program and the new SC model of the transformed program give us equivalent (up to weak bisimulation) transition systems. Furthermore, we aim at generalizing the transformation to a larger class of programs.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Automatic Fence Insertion in Integer Programs via Predicate abstraction. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 164–180. Springer, Heidelberg (2012)
2. Adve, S.V., Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial. IEEE Computer 29(12), 66–76 (1996)
3. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems. LNCS, vol. 7792, pp. 512–532. Springer, Heidelberg (2013)
4. AMD. AMD64 Architecture Programmer's Manual Volume 2: System Programming (2012), `http://support.amd.com/us/Processor_TechDocs/24593_APM_v2.pdf`
5. Arora, N.S., Blumofe, R.D., Greg Plaxton, C.: Thread Scheduling for Multiprogrammed Multiprocessors. In: Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 1998, pp. 119–129. ACM, New York (1998)
6. 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)
7. Batty, M., Dodds, M., Gotsman, A.: Library abstraction for C/C++ concurrency. In: POPL, pp. 235–248 (2013)
8. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and Enforcing Robustness against TSO. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems. LNCS, vol. 7792, pp. 533–553. Springer, Heidelberg (2013)
9. Burckhardt, S., Gotsman, A., Musuvathi, M., Yang, H.: Concurrent Library Correctness on the TSO Memory Model. In: Seidl, H. (ed.) Programming Languages and Systems. LNCS, vol. 7211, pp. 87–107. Springer, Heidelberg (2012)
10. Burnim, J., Sen, K., Stergiou, C.: Sound and complete monitoring of sequential consistency for relaxed memory models. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 11–25. Springer, Heidelberg (2011)
11. Burnim, J., Sen, K., Stergiou, C.: Testing concurrent programs on relaxed memory models. In: Dwyer, M.B., Tip, F. (eds.) ISSTA, pp. 122–132. ACM (2011)
12. Burns, J., Lynch, N.A.: Mutual Exclusion Using Indivisible Reads and Writes. In: Proceedings of the 18th Annual Allerton Conference on Communication, Control, and Computing, pp. 833–842 (1980)

13. 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, 451–490 (1991)
14. Dan, A.M., Meshman, Y., Vechev, M., Yahav, E.: Predicate abstraction for relaxed memory models. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 84–104. Springer, Heidelberg (2013)
15. Derrick, J., Schellhorn, G., Wehrheim, H.: Mechanically verified proof obligations for linearizability. ACM Trans. Program. Lang. Syst. 33(1), 4 (2011)
16. Derrick, J., Smith, G., Dongol, B.: Verifying linearizability on TSO architectures. In: iFM (to appear, 2014)
17. Gotsman, A., Musuvathi, M., Yang, H.: Show no weakness: Sequentially consistent specifications of TSO libraries. In: Aguilera, M.K. (ed.) DISC 2012. LNCS, vol. 7611, pp. 31–45. Springer, Heidelberg (2012)
18. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
19. Intel, Santa Clara, CA, USA. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1 (May 2012)
20. Kuperstein, M., Vechev, M.T., Yahav, E.: Automatic Inference of Memory Fences. SIGACT News 43(2), 108–123 (2012)
21. Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Trans. Computers 28(9), 690–691 (1979)
22. Owens, S.: Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 478–503. Springer, Heidelberg (2010)
23. Corina, S.: Păsăreanu and Willem Visser. A Survey of New trends in Symbolic Execution for Software Testing and Analysis. Int. J. Softw. Tools Technol. Transf. 11(4), 339–353 (2009)
24. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured Specifications and Interactive Proofs with KIV. In: Automated Deduction—A Basis for Applications. Interactive Theorem Proving, vol. II, ch. 1, pp. 13–39. Kluwer (1998)
25. Schellhorn, G., Wehrheim, H., Derrick, J.: How to prove algorithms linearisable. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 243–259. Springer, Heidelberg (2012)
26. 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)
27. 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)
28. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: Torrellas, J., Chatterjee, S. (eds.) PPOPP, pp. 129–136 (2006)
29. Vechev, M., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: Păsăreanu, C.S. (ed.) SPIN 2009. LNCS, vol. 5578, pp. 261–278. Springer, Heidelberg (2009)