Verifying linearizability on TSO architectures

John Derrick[1], Graeme Smith[2], and Brijesh Dongol[1]

[1]Department of Computing, University of Sheffield, Sheffield, UK
[2]School of Information Technology and Electrical Engineering,
The University of Queensland, Australia

Abstract. Linearizability is the standard correctness criterion for fine-grained, non-atomic concurrent algorithms, and a variety of methods for verifying linearizability have been developed. However, most approaches assume a sequentially consistent memory model, which is not always realised in practice. In this paper we define linearizability on a *weak* memory model: the TSO (Total Store Order) memory model, which is implemented in the x86 multicore architecture. We also show how a simulation-based proof method can be adapted to verify linearizability for algorithms running on TSO architectures. We demonstrate our approach on a typical concurrent algorithm, spinlock, and prove it linearizable using our simulation-based approach. Previous approaches to proving linearizabilty on TSO architectures have required a modification to the algorithm's natural abstract specification. Our proof method is the first, to our knowledge, for proving correctness without the need for such modification.

1 Introduction

The correctness of concurrent algorithms has received considerable attention over the last few years. For algorithms that have fine-grained concurrent implementations correctness has focussed on a condition called *linearizability* [12]. This requires that the fine-grained operations (e.g., insertion or removal of an element of a data structure) appear as though they take effect "instantaneously at some point in time within their intervals of execution" [12], thereby achieving the same effect as an atomic operation.

Such fine-grained implementations are becoming increasingly commonplace, and are now standard in libraries such as `java.util.concurrent`. To increase efficiency, these algorithms dispense with locking, or only lock small parts of a shared data structure. Therefore the shared data structure might be concurrently accessed by different processors executing different operations. This complexity makes the correctness of such algorithms, i.e., their proofs of linearizability, a key issue.

Because linearizability is such an important condition, there has been a large amount of interest in proof methods for verifying whether an algorithm is linearizable. However, the vast majority of this work has assumed a particular memory model; in particular a *sequentially consistent* (SC) memory model, whereby program instructions are executed by the hardware in the order specified by the program. This is in contrast to multiprocessor architectures such as x86 [15], Power [1] or ARM [1] that only provide weaker guarantees in order to allow efficient executions.

Processor cores within modern multicore systems often communicate via shared memory and use (local) store buffers to improve performance. Whilst this does give

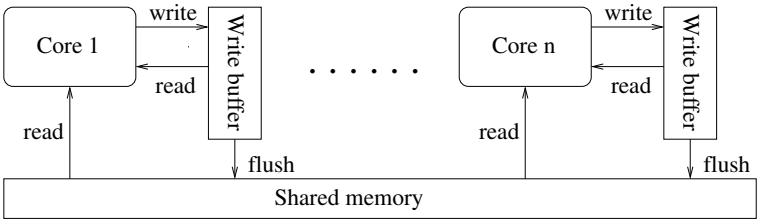

Fig. 1. The TSO architecture

greater scope for optimisation, the order in which instructions are executed by the hardware is no longer the same as that specified by the program. In this paper we focus on one such memory model, the TSO (Total Store Order) model which is implemented in the x86 multicore processor architecture. Rather surprisingly given Intel's and AMD's use of x86, and in contrast to the many different approaches and techniques for linearizability on sequentially consistent architectures, there have only been three approaches to the question of linearizability on a TSO model: [4], [11] and [17].

The proof approaches in both [4] and [11], however, require the natural abstract specification of a concurrent implementation to be modified. Burckhardt et al. [4] define so-called TSO-to-TSO linearizability, which as the name implies compares a concurrent implementation with an abstract specification that executes in TSO memory. Their definition of linearizability thus compares two specifications with local buffers. In our approach, we aim to compare the execution of an implementation in TSO against its natural SC abstraction. In [11] Gotsman et al. define a more sophisticated mapping between the TSO model and a sequentially consistent one (this is called TSO-to-SC linearizability), but to verify the linearizability of an example like spinlock (our running example) they weaken the abstract specification to allow non-deterministic behaviour when one would not naturally expect it. The approach closest to ours is [17] which uses the same principles that we discuss, but does not provide a proof method (rather they use SPIN to model check particular runs of the algorithm, much like testing).

The purpose of this paper is to make two contributions: define linearizability of concurrent algorithms on a TSO memory model which avoids the compromises of [4, 11], and define a proof method for verifying it. We begin in Section 2 by introducing the TSO model as well as our running example, the spinlock algorithm. In Section 3 we introduce linearizability and discuss how we adapt the definition to the TSO model. In Section 4 we explain an existing simulation-based method for verifying linearizability and show how we can adapt this to the TSO model. This method is then applied to the spinlock example in Section 5 before we conclude in Section 6.

2 The TSO memory model

In the TSO (Total Store Order) architecture (see [16] for an introduction), each processor core uses a write buffer (as shown in Figure 1), which is a FIFO queue that stores pending writes to memory. A processor core (from this point on referred to as a *process*) performing a *write* to a memory location enqueues the write to the buffer and continues

```
word x=1;

void acquire()
{
  while(1) {
    lock;
    if (x==1) {
      x=0;
      unlock;
      return;
     }
    unlock;
    while(x==0){};
  }
}

void release()
{
  x=1;
}

int tryacquire()
{
  lock;
  if (x==1) {
    x=0;
    unlock;
    return 1;
  }
  unlock;
  return 0;
}
```

Fig. 2. Spinlock implementation

computation without waiting for the write to be committed to memory. Pending writes do not become visible to other processes until the buffer is *flushed*, which commits (some or all) pending writes to memory.

The value of a memory location *read* by a process is the most recent in the processor's local buffer. If there is no such value (e.g., initially or when all writes corresponding to the location have been flushed), the value of the location is fetched from memory. The use of local buffers allows a read by one process, occurring after a write by another, to return an older value as if it occurred before the write.

In general, flushes are controlled by the CPU. However, a programmer may explicitly include a *fence*, or *memory barrier*, instruction in a program's code to force a flush to occur. Therefore, although TSO allows non-sequentially consistent executions, it is used in many modern architectures on the basis that these can be prevented, where necessary, by programmers using fence instructions.

A pair of *lock* and *unlock* commands in TSO allows a process to acquire sole access to the memory. Both commands include a memory barrier which forces the store buffer of that process to be flushed completely (via a sequence of atomic flushes).

2.1 Example - spinlock

Spinlock [3] is a locking mechanism designed to avoid operating system overhead associated with process scheduling and context switching. A typical implementation of spinlock is shown in Fig. 2, where a global variable `x` represents the lock and is set to 0 when the lock is held by a process, and 1 otherwise. A process trying to acquire the lock `x` *spins*, i.e., waits in a loop and repeatedly checks the lock for availability. It is particularly efficient when processes only spin for short periods of time and is often used in operating system kernels.

The `acquire` operation only terminates if it successfully acquires the lock. It will lock the global memory[1] so that no other process can write to `x`. If, however, another

[1] Locking the global memory using the TSO `lock` command should not be confused with acquiring the lock of this case study by setting `x` to 0.

process has already acquired the lock (i.e., `x==1`) then it will unlock the global memory and spin, i.e., loop in the while-loop until it becomes free, before starting over. Otherwise, it acquires the lock by setting `x` to 0.

The operation `release` releases the lock by setting `x` to 1. We assume that only a process that has acquired the lock will call this operation. The operation `tryacquire` differs from `acquire` in that it only makes one attempt to acquire the lock. If this is successful it returns 1, otherwise it returns 0.

The `lock` and `unlock` commands act as memory barriers. Hence, writes to `x` by the `acquire` and `tryacquire` operations are not delayed. For efficiency, however, `release` does not have a memory barrier and so its write to `x` can be delayed until a flush occurs. This leads to the possibility of a `tryacquire` operation of a process q returning 0 after the lock has been released by another process p. For example, the following concrete execution is possible, where we write $(q, \texttt{tryacquire}(0))$ to denote process q performing a `tryacquire` operation and returning 0, and $\texttt{flush}(p)$ to denote the CPU flushing a value from process p's buffer:

$$\langle (p, \texttt{acquire}), (p, \texttt{release}), (q, \texttt{tryacquire}(0)), \texttt{flush}(p) \rangle \tag{1}$$

Thus p performs an `acquire`, then a `release` and then q performs a `tryacquire` that returns 0 even though it occurs immediately after the `release`. This is because the $\texttt{flush}(p)$, which sets the value of x in memory to 0 has not yet occurred.

At an abstract level, the operations are captured by the following Z specification, which has parameterised operations $Acquire_p$, $Release_p$ and $TryAcquire_p$, the parameter p denoting the identifier of the process performing the operation.

AS

$x : \{0, 1\}$

Init

AS

$x = 1$

$Acquire_p$

ΔAS

$x = 1$

$x' = 0$

$Release_p$

ΔAS

$x = 0$

$x' = 1$

$TryAcquire_p$

ΔAS

$out! : \{0, 1\}$

---

if $x = 1$

then $x' = 0 \land out! = 1$

else $x' = x \land out! = 0$

The question is now: "Is the behaviour of spinlock under TSO comparable to this abstract specification?". We answer this question in the subsequent sections by proving that spinlock under TSO is actually linearizable with respect to the specification.

3 Linearizability on TSO

Linearizability [12] is the standard notion of correctness for concurrent algorithms, and allows one to compare a fine-grained implementation against its abstract specification.

The comparison is made at the level of invocations and returns of operations as the fine-grained nature of some operations means that an operation's steps might be interleaved with steps of another operation executed by another process. For example, process *p* might start a `release`, but then process *q* invokes its `tryacquire` before *p*'s `release` has returned. The key idea of linearizability is as follows.

> Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its return. This point is known as the *linearization point*.

In other words, if two operations overlap, then they may take effect in any order from an abstract perspective, but otherwise they must take effect in program order.

The original definition in [12] (for a formalisation, see [7]) is based on the concept of *possibilities*, however there are now a number of different proof strategies which have been applied to a number of algorithms. These range from using shape analysis [2, 5] and separation logic [5] to rely-guarantee reasoning [18] and refinement-based simulation methods [10, 14, 7]. The simulation-based methods, which we will adapt for use in this paper, show that an abstraction (or simulation or refinement) relation exists between the abstract specification of the data structure and its concurrent implementation.

We will return to the proof method in Section 4. In this section we address the question: Is *spinlock* linearizable on TSO? The definition of linearizability is architecture-neutral, so we should be able to answer the question on a TSO memory model. However, the presence of local buffers, and operations under control of the CPU (i.e., the flushes) complicate the answer.

Consider the execution of spinlock in (1). Obviously, such an execution has no corresponding behaviour at the abstract level, since looking at the Z specification, the value of *x* after *Release* is 1, thus *TryAcquire* returns 1. Hence standard approaches to proving linearizability will fail. There are three alternative approaches to tackling the issue of linearizability on TSO: [4], [11] and [17]. Of these, both [4] and[11] involve changes to the natural abstract specification. For example, in [4] the abstract specification is described with local buffers and flushes. Linearizability, as they define it, (which they call TSO-to-TSO linearizability) then compares two specifications both with local buffers, but this seems to miss the essential nature of the abstract to concrete transformation. On the other hand in [11], Gotsman et al. weaken the abstract specification to allow `tryacquire` to nondeterministically either fail or succeed when `x` is 1, i.e.,

TryAcquire2_p

ΔAS
$out! : \{0, 1\}$

---

if $x = 1$
then $(x' = 0 \land out! = 1) \lor (x' = x \land out! = 0)$
else $x' = x \land out! = 0$

The nondeterminism in the abstract operation models that introduced to the concrete system by hardware-controlled flushes. Since the abstract specification does not have local buffers in it, the authors call it TSO-to-SC linearizability. Again, changing the

abstract specification seems to weaken what one has achieved with the proof. The approach closest to ours is [17] which uses the same principles that we use here, but uses model checking to test linearizability. Our aim is to formalise this intuition and provide a refinement-based proof method for it.

Although at first sight it would seem that linearizability simply fails without changing the abstract specification, one needs to take into account the role of the local buffers. Since the flush of a process's buffer is sometimes the point that the effect of an operation's changes to memory become globally visible, the flush can be viewed as being the final part of the operation. For example, the flush of a variable, such as `x`, after an operation, such as `release`, can be taken as the return of that operation. Under this interpretation, the `release` operation extends from its invocation to the flush which writes its change to `x` to the global memory. The key point is the following principle:

> The return point of an operation on a TSO architecture is not necessarily the point where the operation ceases execution, but can be any point up to the last flush of the variables written by that operation.

Formalisation: We now formalise this intuition. In the standard definition of linearizability, *histories* are sequences of *events* which can be invocations or returns of operations from a set I and performed by a particular process from a set P. Invocations have an associated input from domain *In*, and returns an output from domain *Out* (we assume both domains contain an element $\perp$ denoting no input or output, respectively). On TSO, we generalise events so that they can also be flushes which are performed by the CPU and operate on a particular process's buffer:

$$\mathit{Event} ::= \mathit{inv}\langle\langle P \times I \times \mathit{In}\rangle\rangle \mid \mathit{ret}\langle\langle P \times I \times \mathit{Out}\rangle\rangle \mid \mathit{flush}\langle\langle P\rangle\rangle$$
$$\mathit{History} == \mathrm{seq}\,\mathit{Event}$$

The TSO history corresponding to the execution (1) is[2]:

$$\langle \mathit{inv}(p, \texttt{acquire},), \mathit{ret}(p, \texttt{acquire},), \mathit{inv}(p, \texttt{release},), \mathit{ret}(p, \texttt{release},) \\ \mathit{inv}(q, \texttt{tryaquire},), \mathit{ret}(q, \texttt{tryacquire}, 0), \mathit{flush}(p)\rangle \quad (2)$$

To prove linearizability on a TSO architecture we transform this history to one where the flush on p is the return of the `release`, since in TSO it is the flush that makes the effect visible. The original return of the `release` in the history above is removed. That is, the above history is transformed to:

$$\langle \mathit{inv}(p, \texttt{acquire},), \mathit{ret}(p, \texttt{acquire},), \mathit{inv}(p, \texttt{release},), \\ \mathit{inv}(q, \texttt{tryaquire},), \mathit{ret}(q, \texttt{tryacquire}, 0), \mathit{ret}(p, \texttt{release}),)\rangle \quad (3)$$

In general, we need to transform a history h consisting of invocations, returns and flushes to a history $\mathit{Trans}(h)$ which replaces flushes by the appropriate returns whilst removing all other flushes and returns that are no longer required. The transformation $\mathit{Trans}(h)$ is formalised below. This new history consists just of invocations and returns, the latter indicating when the effect of an operation is made visible globally.

[2] We omit $\perp$ in the events of this and subsequent histories in this section, e.g., $\mathit{inv}(p, \texttt{acquire},)$ denotes $\mathit{inv}(p, \texttt{acquire}, \perp)$.

3.1 Defining the transformation

We first present a number of preliminary definitions that enable the transformation to be carried out deterministically. Let $mp(p, m, n, h)$ denote matching pairs of invocations and returns by process p in history h as in [7]. Its definition requires that $h(m)$ and $h(n)$ are executed by the same process p and are an invocation and return event, respectively, of the same operation. Additionally, it requires that for all k between m and n, $h(k)$ is not an invocation or return event of p. That is, given $inv?(e)$ and $ret?(e)$ denote that the event e is an invocation and return event, respectively, $e.\pi$ denotes the process executing e, and $e.i$ the operation being executed, $mp(p, m, n, h)$ holds iff

$$
\begin{array}{l}
0 < m < n \leq \#h \wedge \\
inv?(h(m)) \wedge ret?(h(n)) \wedge h(m).\pi = h(n).\pi = p \wedge h(m).i = h(n).i \wedge \\
\forall\, k \bullet m < k < n \Rightarrow h(k).\pi \neq p
\end{array}
$$

Let $bs(p, m, h)$ denote the size of process p's buffer at point m in the history h, and $nf(p, m, n, h)$ denote the number of flushes of process p's buffer between points m and n in h. The number of new items in process p's buffer between two points m and n in a history h is given by

$$bi(p, m, n, h) \mathrel{\widehat{=}} bs(p, n, h) + nf(p, m, n, h) - bs(p, m, h)$$

We use the function mpf below to find indices m, n and l in h such that (m, n) is a matching pair and l corresponds to the point to which the return of the matching pair must be moved.

$$
\begin{array}{l}
mpf(p, m, n, l, h) \mathrel{\widehat{=}} mp(p, m, n, h) \wedge n \leq l \wedge \\
\qquad\qquad \mathsf{if}\ bi(p, m, n, h) = 0\ \mathsf{then}\ l = n \\
\qquad\qquad \mathsf{else}\ h(l) = flush(p) \wedge \\
\qquad\qquad\qquad nf(p, m, l, h) = bs(p, m, h) + bi(p, n, m, h)
\end{array}
$$

The first part of the if states that $l = n$ if no items are put on the buffer by the operation invoked at point m. The second states that l corresponds to a flush of p's buffer and the number of flushes between m and l is precisely the number required to flush the contents of the buffer at m and any items added to the buffer between m and n.

To transform a history h, we do the following two steps.

Step 1. Given $mpf(p, m, n, l, h)$ holds for some p:

if $n \neq l$
then $h(l)$ becomes $h(n)$ and $h(n)$ becomes a dummy event δ
else we do nothing because the return should not be moved.

This results in a history $h' \in \text{seq}(Event \cup \{\delta\})$, where all returns have been moved to the return positions of their corresponding flushes.

Step 2. The second step is straightforward: all δ and flushes are removed.

The algorithm described above is deterministic; we let $Trans(h)$ be the function that returns a transformed history by applying the algorithm to history h. For example, history (2) is transformed to the following via Step 1

$$\langle inv(p, \texttt{acquire},), ret(p, \texttt{acquire},), inv(p, \texttt{release},), \delta,$$
$$inv(q, \texttt{tryaquire},), ret(q, \texttt{tryacquire}, 0), ret(p, \texttt{release},)\rangle$$

which in turn is transformed to history (3) by Step 2.

3.2 TSO linearizability

A formal definition of linearizability is given in [7]. We adapt this definition as follows. An incomplete history h is extended with a sequence h_0 of return and flush events, then matched to a sequential history hs by removing the remaining pending invocations using a function *complete*, i.e., $complete(h)$ is a subhistory of h formed by removing all pending invocations from h. We say a history h is *legal* iff for each $n : 1..\#h$ such that $ret?(h(n))$, there exists an earlier $m : 1..n-1$ such that $mp(p, m, n, h)$, and for each $n : 1..\#h$ such that $h(n) = flush(p)$, $bs(p, n, h) > 0$.

A key part of adapting the standard definition to TSO is what we mean by a matching pair of invocations and returns. The formal definition of the function mp in [7] (defined above) requires that for all k between m and n, $h(k)$ is not an invocation or return event of p. This is not true for our transformed histories on TSO since operations by the same process may overlap. Therefore, we will use a new version of matching pairs mp_{TSO} defined as follows.

$$mp_{TSO}(p, m, n, h) \text{ iff } mpf(p, x, z, y, h)$$
$$\text{where } m = x - \sum_{p:P} nf(p, 1, x, h) \text{ and } n = y - \sum_{p:P} nf(p, 1, y, h) \text{ and } x < z \leq y$$

Given RF is the set of all return and flush events, we define TSO linearizability as follows.

Definition 1 (TSO linearizability). *A history h : History is* TSO linearizable *with respect to some sequential history hs iff $lin(h, hs)$ holds, where*

$$lin(h, hs) \mathrel{\widehat{=}} \exists\, h_0 : seq\, RF \bullet legal(h \frown h_0) \wedge linrel(Trans(complete(h \frown h_0)), hs)$$

where

$$\begin{aligned} linrel(h, hs) \mathrel{\widehat{=}} \exists f : 1..\#h \rightarrowtail 1..\#hs \bullet\ & (\forall n : 1..\#h \bullet h(n) = hs(f(n))) \wedge \\ & (\forall p : P;\ m, n : 1..\#h \bullet m < n \wedge mp_{TSO}(p, m, n, h) \Rightarrow f(n) = f(m) + 1) \wedge \\ & (\forall p, q : P;\ m, n, m', n' : 1..\#h \bullet \\ & \quad n < m' \wedge mp_{TSO}(p, m, n, h) \wedge mp_{TSO}(q, m', n', h) \Rightarrow f(n) < f(m')) \quad \square \end{aligned}$$

That is, operations in hs do not overlap (each invocation is followed immediately by its matching return) and the order of non-overlapping operations in h is preserved in hs.

Note that history (2) is a complete legal history, and that *Trans* applied to this history gives us history (3). Since `release` and `tryacquire` now overlap in the transformed history (3), a potential linearization in terms of the abstract specification is

$$\langle inv(p, Acquire,), ret(p, Acquire,), inv(q, TryAcquire,), ret(q, TryAcquire, 0),$$
$$inv(p, Release,), ret(p, Release,)\rangle$$

Thus, spinlock is TSO linearizable with respect to the abstract specification.

4 A proof method for linearizability on TSO

We do not work directly with this definition of linearizability, but rather use a refinement-based proof method for verifying linearizability as defined in [6–8, 14]. This approach defines simulation rules that form a sound (and complete) proof method for verifying linearizability. Different classes of algorithm use slightly different rules, where the difference depends on how easy it is to identify the linearization points — in some algorithms these can't be identified directly, and depend on the behaviour of other processes [8]. However, for the example in this paper we can use the simplest set of rules found in [7] and described below.

General approach. The approach is based on proving a concrete specification that has one operation for each line of code is a *non-atomic refinement* [9] of the abstract specification capturing the code's intent. Each allowable sequence of concrete steps must simulate a sequence of abstract operations despite the interleaving of concrete steps performed by different processes[3].

Let P be the set of processes. Let our abstract and concrete specifications be given as $A = (AState, AInit, (AOP_{p,i})_{p\in P, i\in I})$ and $C = (CState, CInit, (COP_{p,j})_{p\in P, j\in J})$ where the sets I and J are used to index the abstract operations and concrete steps, respectively. The function $abs : J \to I$ maps each concrete step to the abstract operation it (together with other steps) implements. We assume the concrete state space *CState* is composed of a global state *GS* (the shared memory) and the local state *LS* of one process (the program counter, local variables and, on TSO, the local buffer). Following [7], linearizability is then shown by:

1. *Defining a* status *function that identifies the linearization points of operations.*
   Let $STATUS ::= IDLE \mid IN\langle\langle In\rangle\rangle \mid OUT\langle\langle Out\rangle\rangle$ where *In* and *Out* are the domains of inputs and outputs, respectively, as defined in Section 3. We define a function $status : GS \times LS \to STATUS$ such that the following hold.
   If a process has no pending operation then the status of the process is *IDLE*. If it is executing an operation and has not passed the linearization point, then the status of the process is $IN(in)$ where *in* is the input of the operation, if any, and $\perp$ otherwise. If it is executing an operation and has passed the linearization point, the status is $OUT(out)$ where *out* is the output of the operation if any, and $\perp$ otherwise.
2. *Showing individual concrete runs of a process correctly implement the abstract operations using non-atomic refinement.*
   We find a forward simulation R relating the global state and the local state of a process to the abstract state, i.e., $R \subseteq AState \times (GS \times LS)$, and a set of simulation rules which additionally update the status function appropriately as shown in the example in Fig. 3. In this example, the input *in* of an invocation step $INVOP(in)$ is used to establish a status of $IN(in)$. After the invocation an internal operation implements *skip* and leave the status unchanged. Then the linearization point that implements $AOp(in, out)$ is passed and the status changes to $OUT(out)$. Finally the status is used to compute the output of a return step $RETOP(out)$ and the status returns to *IDLE*.

[3] We use the term *steps* in this section to distinguish the concrete operations of the specification from the operations of the code (such as `acquire` in our example).

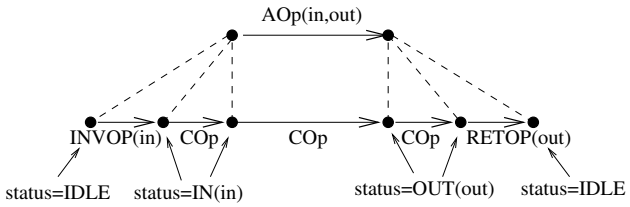

Fig. 3. The status information for non-atomic refinement

There are 5 different simulation rules depending on whether the particular concrete step being considered is an invocation step, a return step, or an internal step before linearization, after linearization or at the linearization point. As an example, the simulation rule for a concrete invocation step is[4]:

$$\begin{array}{l}\forall\, as : AState;\ gs, gs' : GS;\ ls, ls' : LS;\ in : In \bullet \\ \quad R(as, gs, ls) \wedge status(gs, ls) = IDLE \wedge INVOP_j(in, gs, ls, gs', ls') \\ \quad\quad \Rightarrow (status(gs', ls') = IN(in) \wedge R(as, gs', ls')) \\ \quad\quad\quad \vee (\exists\, as' : AState;\ out : Out \bullet \\ \quad\quad\quad\quad AOP_{abs(j)}(in, as, as', out) \wedge status(gs', ls') = OUT(out) \wedge \\ \quad\quad\quad\quad R(as', gs', ls'))\end{array}$$

where the first and second disjuncts in the consequent capture invocations that do and do not correspond to a linearization, respectively.

3. *Showing* interference freedom*, i.e., that other processes running in parallel do not destroy this non-atomic refinement.*
   To ensure steps of other processes preserve the local simulation relation R, we define $R(as, gs, ls) \mathrel{\widehat{=}} ABS(as, gs) \wedge INV(gs, ls)$ where $ABS(as, gs)$ captures how the abstract state is represented by the global state, and $INV(gs, ls)$ provides further constraints between the global and local variables. $ABS(as, gs)$ is preserved by all steps of *all* processes. Hence it is sufficient to prove that $INV(gs, ls)$ is preserved by other processes. The *interference freedom* condition is:

$$\begin{array}{l}\forall\, as : AState;\ gs, gs' : GS;\ ls, ls', lsq : LS \bullet \\ \quad ABS(as, gs) \wedge INV(gs, ls) \wedge INV(gs, lsq) \wedge D(ls, lsq) \wedge COP_j(gs, ls, gs', ls') \\ \quad\quad \Rightarrow INV(gs', lsq) \wedge D(ls', lsq) \wedge status(gs', lsq) = status(gs, lsq)\end{array}$$

   where a symmetric predicate $D \subseteq LS \times LS$ is used to constrain the relationship between the local states of any two processes. This predicate must also be preserved by the steps of all processes.

4. *Showing the concrete initialisation satisfies the abstract initialisation.*

$$\begin{array}{l}\forall\, gs : GSInit \bullet \exists\, as : AInit \bullet \\ \quad ABS(as, gs) \wedge (\forall\, ls : LSInit \bullet INV(gs, ls)) \wedge (\forall\, ls, lsq : LSInit \bullet D(ls, lsq))\end{array}$$

[4] In this paper, we use $R(x, y)$ and $R(x, y, z)$ as shorthands for $(x, y) \in R$ and $(x, (y, z)) \in R$, respectively, for all relations R.

where *GSInit* and *LSInit* are the initial states of the global and local state spaces, respectively. □

The *status* function captures the status of a single pending operation. Under TSO, however, we may have several pending operations: that operation currently being executed by the process, if any, and those that have completed apart from the flushing of their writes to memory. In our extension to the above approach, we let *status* capture the status of the operation which the process is currently executing. If there is no such process, the status is *IDLE*. All other pending operations, i.e., those completed apart from flushes, will necessarily be before their linearization points.

The other role of the *status* function is to carry the inputs of the operation until they are needed at the linearization point where they, along with the operation's outputs, must match those of the associated abstract operation. To adapt the approach to TSO, we need to be able to keep track of the inputs of completed, but pending, operations. We also need to keep track of the abstract operation associated with each completed, but pending, operation and the completed operation's outputs (since the operation has already completed, its outputs will have already occurred).

To do this we add four auxiliary variables to the local concrete state space *LS*. The first of these $lin : \text{seq}((I \cup \{null\}) \times In \times Out)$ records, for each buffer entry, the abstract operation for which its flush is a linearization point (*null* indicates its flush is not a linearization point), the abstract operation's input, and the abstract operation's output. When a flush occurs the values corresponding to the flushed entry are read into the other three auxiliary variables $op : I \cup \{null\}$, $in : In$ and $out : Out$.

We also need to introduce two new simulation rules. The first corresponds to a process with status *IN* returning to *IDLE* without linearizing. This would be the case where the operation is to be linearized by a flush which is yet to occur.

Return without Lin.

$$\forall as : AState;\ gs, gs' : GS;\ ls, ls' : LS;\ in : In \bullet$$
$$R(as, gs, ls) \wedge status(gs, ls) = IN(in) \wedge RETOP_j(gs, ls, gs', ls', out) \Rightarrow$$
$$status(gs', ls') = IDLE \wedge R(as, gs', ls')$$

The second corresponds to the occurrence of a flush. A flush acts as either an internal step or a linearizing step as shown in cases (a) and (b) of Fig. 4, respectively. Case (a) can occur when the process has any status and its status is not changed. When the status is *IN* or *OUT* the internal step may be of the pending operations, and for any status it may be of an operation which has previously completed.

Case (b) also occurs from any status and when the status is *IDLE* or *OUT* it remains unchanged. Such a flush in these statuses corresponds to the linearization of an operation which has already completed. When the status is *IN* the flush may also be linearizing an operation which has already completed, in which case the status is unchanged, or it may be linearizing the pending operation, in which case the status becomes *OUT*.

The rule refers to the post-states of the auxiliary variables *op*, *in* and *out* via $ls'.op$, $ls'.in$ and $ls'.out$, respectively.

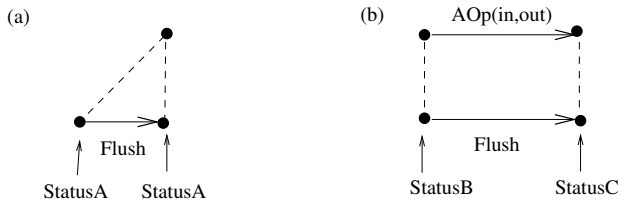

Fig. 4. Simulation rules for flush

Flush.

$$\forall as : AState;\ gs, gs' : GS;\ ls, ls' : LS;\ in : In \bullet$$
$$R(as, gs, ls) \land Flush(gs, ls, gs', ls') \Rightarrow$$
$$(ls'.op = null \Rightarrow R(as, gs', ls') \land status(gs', ls') = status(gs, ls)) \land$$
$$(ls'.op \neq null \Rightarrow$$
$$(\exists as' : AState \bullet AOP_{ls'.op}(ls'.in, as, as', ls'.out) \land R(as', gs', ls')) \land$$
$$(status(gs', ls') = status(gs, ls)$$
$$\lor (status(gs, ls) = IN(ls'.in) \land status(gs', ls') = OUT(ls'.out)))$$

5 Spinlock is linearizable on TSO

To show that spinlock is linearizable using the approach described in Section 4, we produce a concrete specification of the algorithm. Given P is the set of all process identifiers, the global state of the concrete specification includes the value of the shared variable x which is initially 1, and a variable *lock* denoting which process, if any, currently has the global memory locked.

GS

$x : \{0, 1\}$
$lock : \mathbb{P}\, P$

$\#lock \leq 1$

GSInit

GS

$x = 1$
$lock = \varnothing$

The local state of a given process is specified in terms of its process identifier from P, a program counter indicating which operation (i.e., line of code) can next be performed, and the process's buffer.

Let $PC ::= 1 \mid 2 \mid a1 \mid \ldots \mid a8 \mid ta1 \mid \ldots \mid ta7 \mid r1$ where the value 1 denotes the process is idle when it has not acquired the spinlock, the value 2 denotes the process is idle when it *has* acquired the spinlock, the values ai, for $i \in 1 \,..\, 8$, denote the process is ready to perform the ith line of code of `acquire`, the values tai, for $i \in 1 \,..\, 7$, denote the process is ready to perform the ith line of code of `tryacquire`, and the value $r1$ denotes the process is ready to perform the first line of `release`.

LS_0

$id : P$
$pc : PC$
$buffer : \text{seq}\{0, 1\}$

As detailed in Section 4, we add auxiliary variables to our local state to keep track of information required at linearization points corresponding to a flush. Let $I == \{1, 2, 3\}$ be the indices of the abstract operations such that 1 denotes *Acquire*, 2 denotes *Release* and 3 denotes *TryAcquire*. Let $In == \{\bot\}$ be the set of input values of operations, and $Out == \{0, 1, \bot\}$ be the set of output values.

LS

LS_0
$op : I \cup \{null\}$
$in : In$
$out : Out$
$lin : \text{seq}((I \cup \{null\}) \times In \times Out)$

$\#lin = \#buffer$

LSInit

LS

$pc = 1$
$buffer = \langle\,\rangle$

Given this specification, the lines of code are formalised as Z operations[5]. For example, for the `acquire` operation we have an operation $A0$ corresponding to the invocation of the operation, an operation $A1$ corresponding to the line of code `while(1)`, and an operation $A2$ corresponding to the line of code `lock`.

$A0$

ΞGS
ΔLS

$pc = 1$
$pc' = a1$

$A1$

ΞGS
ΔLS

$pc = a1$
$pc' = a2$

$A2$

ΔGS
ΔLS

$lock = \varnothing \land pc = a2$
$lock' = \{id\} \land pc' = a3$

To model the fact that $A2$ also results in all entries of the process's buffer being flushed, the operation $A3$ corresponding to the following line of code, `x=1`, is not enabled unless $buffer = \langle\,\rangle$. It will become enabled after the required number of *Flush* operations have occurred. These remove an entry from the buffer and update the auxiliary variables *op*, *in* and *out* according to the information in *lin*. This information is added to *lin* when the buffer entries are added. For example, the operation $A4$, corresponding to the line `x=0`, updates *lin* to indicate that the flush of this value will not be a linearization point.

$A3$

ΞGS
ΔLS

$buffer = \langle\,\rangle$
$pc = a3$
if $x = 1$
then $pc' = a4$
else $pc' = a7$

Flush

ΔGS
ΔLS

$lock = \varnothing \lor lock = \{id\}$
$buffer \neq \langle\,\rangle$
$x' = head\ buffer$
$buffer' = tail\ buffer$
$head\ lin = (op', in', out')$
$lin' = tail\ lin$

$A4$

ΞGS
ΔLS

$pc = a4$
$buffer' = buffer \frown \langle 0 \rangle$
$pc' = a5$
$lin' = lin \frown \langle (null, \bot, \bot) \rangle$

[5] To simplify the presentation we adopt the convention that the values of variables that are not explicitly changed by an operation remain unchanged.

The other concrete operations are modelled similarly. The operations corresponding to the `unlock` statements are only enabled when $\mathit{buffer} = \langle\,\rangle$ modelling that the buffer must be completely flushed before the memory is unlocked.

Such a concrete specification is well-formed only if any sequence of operations corresponding to an abstract operation has exactly one linearization point. It is important, therefore, when modelling operations which change *lin* (i.e., those that write to the buffer) to ensure this. If a buffer entry is marked as a linearization point, the sequence of operations in which it occurs should not be linearized by a change in status from *IN* to *OUT*, nor by any other buffer entry.

Linearization can now be proved by defining the function *status* so that the linearization points of *Acquire* and *TryAcquire* are the operations which release the memory lock, and the linearization point of *Release* is the flush that commits the associated value of 1 to the global variable x.

The required relations *ABS*, *INV* and *D* are as follows.

$$\mathit{ABS} : \mathit{AS} \leftrightarrow \mathit{GS}$$

$$\begin{array}{l}\forall\, \mathit{as} : \mathit{AS};\ \mathit{gs} : \mathit{GS} \mid \mathit{ABS}(\mathit{as}, \mathit{gs}) \bullet \\ \quad (\mathit{gs.lock} = \varnothing \Rightarrow \mathit{gs.x} = \mathit{as.x}) \land (\mathit{gs.lock} \neq \varnothing \land \mathit{gs.x} = 1 \Rightarrow \mathit{as.x} = 1)\end{array}$$

$$\mathit{INV} : \mathit{GS} \leftrightarrow \mathit{LS}$$

$$\begin{array}{l}\forall\, \mathit{gs} : \mathit{GS};\ \mathit{ls} : \mathit{LS} \mid \mathit{INV}(\mathit{gs}, \mathit{ls}) \bullet \\ \quad (\mathit{ls.pc} = \{1, a1, a2, a3, ta1, ta2\} \land \mathit{ls.buffer} \neq \langle\,\rangle \Rightarrow \mathit{gs.x} = 0) \land \\ \quad (\mathit{ls.pc} \in \{a4, ta3\} \Rightarrow \mathit{gs.x} = 1) \land \\ \quad (\mathit{ls.pc} \in \{a5, ta4\} \land \mathit{ls.buffer} = \langle\,\rangle \Rightarrow \mathit{gs.x} = 0) \land \\ \quad (\mathit{ls.pc} \in \{a5, ta4\} \land \mathit{ls.buffer} \neq \langle\,\rangle \Rightarrow \\ \qquad \mathit{gs.x} = 1 \land \mathit{ls.buffer} = \langle 0\rangle \land (\mathit{head}\, \mathit{ls.lin}).1 = \mathit{null}) \land \\ \quad (\mathit{ls.pc} \in \{2, a6, ta3, ta6, r1\} \Rightarrow \mathit{gs.x} = 0) \land \\ \quad (\mathit{ls.pc} \notin \{a6, ta6\} \land \mathit{ls.buffer} \neq \langle\,\rangle \Rightarrow \\ \qquad \mathit{ls.buffer} = \langle 1\rangle \land (\mathit{head}\, \mathit{ls.lin}).1 = 2) \land \\ \quad (\mathit{ls.pc} \in \{2, a4, a6, a7, a8, ta3, ta5, ta6, ta7, r1\} \Rightarrow \mathit{ls.buffer} = \langle\,\rangle) \land \\ \quad (\mathit{ls.pc} \in \{a3, a4, ta2, ta3\} \Rightarrow \mathit{gs.lock} = \{\mathit{ls.id}\}) \land \\ \quad (\mathit{gs.x} = 1 \land \mathit{ls.buffer} \neq \langle\,\rangle \Rightarrow \mathit{gs.lock} = \{\mathit{ls.id}\})\end{array}$$

$$D : \mathit{LS} \leftrightarrow \mathit{LS}$$

$$\begin{array}{l}\forall\, \mathit{ls}, \mathit{lsq} : \mathit{LS} \mid D(\mathit{ls}, \mathit{lsq}) \bullet \\ \quad \mathit{ls.buffer} \neq \langle\,\rangle \Rightarrow \mathit{lsq.buffer} = \langle\,\rangle \land \\ \quad \mathit{lsq.buffer} \neq \langle\,\rangle \Rightarrow \mathit{ls.buffer} = \langle\,\rangle \land \\ \quad \mathit{ls.pc} \in \{2, a4, a6, ta3, ta5, r1\} \Rightarrow \mathit{ls.buffer} = \mathit{lsq.buffer} = \langle\,\rangle \land \\ \quad \mathit{lsq.pc} \in \{2, a4, a6, ta3, ta5, r1\} \Rightarrow \mathit{ls.buffer} = \mathit{lsq.buffer} = \langle\,\rangle \land \\ \quad \mathit{ls.pc} \in \{a5, ta4\} \Rightarrow \mathit{lsq.buffer} = \langle\,\rangle \land \\ \quad \mathit{lsq.pc} \in \{a5, ta4\} \Rightarrow \mathit{ls.buffer} = \langle\,\rangle\end{array}$$

For example, consider a *Flush* operation occurring when $\mathit{pc} = a5$. For the operation to occur, $\mathit{buffer} \neq \langle\,\rangle$. Hence, by $\mathit{INV}(\mathit{gs}, \mathit{ls})$, $\mathit{gs.x} = 1$, $\mathit{ls.buffer} = \langle 0\rangle$, $(\mathit{head}\, \mathit{ls.lin}).1 = \mathit{null}$ and $\mathit{gs.lock} = \{\mathit{ls.id}\}$.

Consider the Flush rule of Section 4. Since $ls'.op = (head\ ls.lin).1 = null$ and $status(gs, ls) = IN(\bot)$, this will hold if both $ABS(as, gs')$ and $INV(gs', ls')$ hold and $status(gs', ls') = IN(\bot)$. The latter follows since the operation does not change $ls.pc$. $ABS(as, gs')$ also holds since the operation does not change $gs.lock$ (i.e., $gs'.lock \neq \varnothing$) and sets $gs'.x = 0$. $INV(gs', ls')$ holds since in addition to setting $gs'.x = 0$ the *Flush* operation sets $ls'.buffer = tail\ ls.buffer = \langle\,\rangle$.

We also need to prove non-interference for this operation. Let ls denote the state of the process on whose buffer the flush is performed. Since $ls.buffer \neq \langle\,\rangle$, when $D(ls, lsq)$ holds for all $lsq \neq ls$, all other process buffers are empty. Since the other process's buffers are not changed by the operation, $DS(ls', lsq)$ holds. Although the operation changes the value of the global variable $gs.x$ to 0, $INV(gs', lsq)$ will remain true since it can only be affected by this change when $lsq.pc \in \{a4, ta3, a5, ta4\}$ and in each of these cases $ls.buffer$ would be equal to $\langle\,\rangle$ by $D(ls, lsq)$, i.e., the *Flush* operation would not be enabled.

Since similar proofs can be carried out for each concrete operation, and the initialisation condition holds (since $as.x = gs.x = 1$ implies $ABS(as, gs)$, $ls.pc = 1$ and $ls.buffer = \langle\,\rangle$ implies $INV(gs, ls)$, and $ls.buffer = lsq.buffer = \langle\,\rangle$ implies $D(ls, lsq)$), spinlock is linearizable on TSO.

6 Conclusions

This paper has presented a definition and simulation-based proof method for linearizability on the TSO memory model. The key to our definition is the treatment of flushes of local buffer entries as part of the operation which made the entries. This enables a proof method which, unlike existing methods, can be used to show implementations of algorithms are linearizable with respect to their natural abstract specifications. This work has applied state-based methods to program verification, and is part of a larger effort on the verification of linearizability on sequentially consistent architectures as well as weaker memory models. This larger effort mechanises the proofs of linearizability by integrating the state-based reasoning into the KIV theorem prover, see [6–8], and in [14] we prove (and mechanise the proof) that our approach is complete, in that all linearizable algorithms can be verified by such simulation-based methods. Given we are using the same simulation-based approach, mechanisation and integration into KIV of the theory in this paper will be relatively straightforward.

One interesting consequence of our approach is that operations on a single process may overlap and hence be reordered under linearizability. This is in contrast to earlier work on sequentially consistent architectures where linearizability implies the additional correctness criterion of *sequential consistency* [13], i.e., that operations on a single process occur in the order that they are called. Sequential consistency will only hold on TSO when, in addition to linearizability, memory barriers are included in all operations which (a) do not write to memory, but (b) occur in a process with other operations which do write to memory. This can be checked by inspection of the code.

Investigating alternative definitions of linearizability on TSO which maintain sequential consistency is an area of future work. Other areas of future work include the reducing the effort needed to apply the proof method. For example, the use of a coarse-

grained abstraction as an intermediate layer between the concrete and abstract specifications, and a means to automatically generate the required invariants.

References

1. J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F.Z. Nardelli. The Semantics of Power and ARM Multiprocessor Machine Code. In L. Petersen and M.M.T. Chakravarty, editors, *DAMP '09*, pages 13–24. ACM, 2008.
2. D. Amit, N. Rinetzky, T.W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In W. Damm and H. Hermanns, editors, *CAV 2007*, volume 4590 of *LNCS*, pages 477–490. Springer, 2007.
3. D. Bovet and M. Cesati. *Understanding the Linux Kernel*. OReilly, 3rd edition, 2005.
4. S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In H. Seidl, editor, *ESOP 2012*, volume 7211 of *LNCS*, pages 87–107. Springer, 2012.
5. C. Calcagno, M. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In H.R. Nielson and G. Filé, editors, *SAS 2007*, volume 4634 of *LNCS*, pages 233–238. Springer, 2007.
6. J. Derrick, G. Schellhorn, and H. Wehrheim. Proving linearizability via non-atomic refinement. In J. Davies and J. Gibbons, editors, *IFM 2007*, volume 4591 of *LNCS*, pages 195–214. Springer, 2007.
7. J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.*, 33(1):4, 2011.
8. J. Derrick, G. Schellhorn, and H. Wehrheim. Verifying linearisabilty with potential linearisation points. In M. Butler and W. Schulte, editors, *FM 2011*, volume 6664 of *LNCS*, pages 323–337. Springer, 2011.
9. J. Derrick and H. Wehrheim. Non-atomic refinement in Z and CSP. In *ZB2005*, LNCS. Springer, 2005.
10. S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In D. de Frutos-Escrig and M. Nunez, editors, *FORTE 2004*, volume 3235 of *LNCS*, pages 97–114. Springer, 2004.
11. A. Gotsman, M. Musuvathi, and H. Yang. Show no weakness: Sequentially consistent specifications of TSO libraries. In M. Aguilera, editor, *DISC 2012*, volume 7611 of *LNCS*, pages 31–45. Springer, 2012.
12. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
13. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.
14. G. Schellhorn, H. Wehrheim, and J. Derrick. How to prove algorithms linearisable. In P. Madhusudan and S.A. Seshia, editors, *CAV 2012*, volume 7358 of *LNCS*, pages 243–259. Springer, 2012.
15. P. Sewell, S. Sarkar, S. Owens, F.Z. Nardelli, and M.O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.
16. D.J. Sorin, M.D. Hill, and D.A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.
17. O. Travkin, A. Mütze, and H. Wehrheim. SPIN as a linearizability checker under weak memory models. In V. Bertacco and A. Legay, editors, *HVC'13*, volume 8244 of *LNCS*, pages 311–326. Springer, 2013.
18. V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.