Admit Your Weakness: Verifying Correctness on TSO Architectures

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

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

Abstract. Linearizability has become the standard correctness criterion for fine-grained non-atomic concurrent algorithms, however, most approaches assume a sequentially consistent memory model, which is not always realised in practice. In this paper we study the correctness of concurrent algorithms on a *weak* memory model: the TSO (Total Store Order) memory model, which is commonly implemented by multicore architectures. Here, linearizability is often too strict, and hence, we prove a weaker criterion, *quiescent consistency* instead. Like linearizability, quiescent consistency is compositional making it an ideal correctness criterion in a component-based context. We demonstrate how to model a typical concurrent algorithm, *seqlock*, and prove it quiescent consistent using a simulation-based approach. Previous approaches to proving correctness on TSO architectures have been based on linearizabilty which makes it necessary to modify the algorithm's high-level requirements. Our approach is the first, to our knowledge, for proving correctness without the need for such a modification.

1 Introduction

This paper is concerned with correctness of concurrent algorithms that typically arise in the multicore processor context, in which shared variables or data structures such as queues, stacks or hashtables are accessed concurrently by several processes. These are becoming prevalent in libraries such as `java.util.concurrent` and operating system kernels. To increase efficiency, these algorithms dispense with locking, or only lock small parts of a data structure. Thus the shared variables or data structure might be concurrently accessed by different processors executing different operations — correctness of such algorithms is therefore a key issue.

To date, the subject of correctness has focussed on a condition called *linearizability* [11]. This requires that fine-grained implementations of access operations (e.g., reading or writing of a shared variable) appear as though they take effect instantaneously at some point in time within the operation's interval of execution — thereby achieving the same effect as an atomic operation. There has been an enormous amount of interest in deriving techniques for verifying linearizability. These range from using shape analysis [1, 4] and separation logic [4] to rely-guarantee reasoning [21] and refinement-based simulation methods [8, 6].

The vast majority of this work has assumed a *sequentially consistent* memory model, whereby program instructions are executed by the hardware in the order specified by the program [14]. However, processor cores within modern multicore systems often communicate via shared memory and use (local) store buffers to improve performance. Whilst these *weak* memory models give greater scope for optimisation, sequential consistency is lost (as the effect of a *write* to the shared memory is delayed by the local buffer). One such memory model is the TSO (Total Store Order) model which is implemented in the x86 multicore processor architecture [19].

The purpose of this paper is to investigate correctness of concurrent algorithms in the TSO memory model. There has been limited work in this area so far, with current approaches [3, 9] based on linearizabilty, which makes it necessary to modify the algorithm's high-level requirements. Instead, we focus here on the weaker notion of *quiescent consistency* as a correctness criterion. Like linearizability, quiescent consistency is compositional making it an ideal correctness criterion in a component-based context. Quiescent consistency was introduced in [2, 18] and has been advocated recently by Shavit as the correctness condition to be used in the multicore age [17]. Although these papers provide neither a formal definition nor a proof method for quiescent consistency, both these shortcomings were addressed in [5], for sequentially consistent architectures.

Like linearizability [11], the definition in [5] is formalised in terms of histories of invocations and responses of the operations of the concurrent algorithm, while the proof method is based on coupled simulations [7] of history-enhanced data types [6]. However, the methods in [5] only address concurrent data structures that are designed to be quiescent consistent and execute under sequentially consistent memory. The aim of this paper is to investigate the use of quiescent consistency as the correctness requirement under TSO memory, then adapt the proof method in [5] to verify such algorithms. We are not proposing that quiescent consistency is the definitive correctness criterion for TSO, but rather that it is an alternative to linearizability that may be useful in some circumstances. We illustrate this with our running example in the paper.

We make three contributions. First, we show how we can adapt the definition of quiescent consistency to a TSO memory model (Section 2). Second, we show in Section 3 how we can use Z to model algorithms on a TSO architecture, then in Section 4 using this model we show how we can adapt the simulation-based proof method to verify quiescent consistency. We conclude in Section 5.

2 Background

2.1 The TSO Memory Model

In the TSO (Total Store Order) architecture (see [19] for a good introduction), each processor core uses a write buffer, which is a FIFO queue that stores pending writes to memory. A processor core performing a *write* to a memory location enqueues the write to the buffer and continues computation without waiting for the write to be committed to memory. Pending writes do not become visible to other cores 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 value for that location 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 some 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. To model concurrent algorithms on TSO we assume the following behaviour, which is reflected our Z models [20]: [1]

- A *write* operation to a memory location adds the entry to the tail of its store buffer.
- The head of the store buffer is *flushed* into the memory. This flush is under the control of the CPU and thus happens non-deterministically.
- A *read* of a memory location takes as its value the most recent value in the store buffer if available, and the value from memory otherwise.

Example 1. Consider the following example, with two global variables `x` and `y` which are both initially 0, and operations to write to and read from the variables.

```
word x=0, y=0;

set_x(in word d) {x=d;}     set_y(in word d) {y=d;}
read_x(out word d) {d=x;}   read_y(out word d) {d=y;}
```

A possible execution in TSO is the following sequence of operation calls and flushes:

$$s_1 = \langle (p, \texttt{set\_x}(1)), (p, \texttt{read\_y}(0)), (q, \texttt{set\_y}(1)), (q, \texttt{read\_x}(0)), \\ \texttt{flush}(p), \texttt{flush}(q) \rangle$$

where $(p, \texttt{read\_y}(0))$, for example, means that process p performs a `read_y` operation that returns 0, and $\texttt{flush}(p)$ corresponds to a CPU flush of a single value from p's buffer. Note that both reads return 0, which is not possible on a sequentially consistent architecture. This is because the corresponding *set* operations write to the process's *local* buffer, and these writes are not globally visible until that process's buffer is flushed. Here that happens at the end of the execution. □

2.2 Concurrent Consistency Models

In what sense is a concurrent algorithm correct? Not only do we have executions as in the example above, but the fine-grained nature of operations means that processes do not perform the whole operation at once — an operation's steps might be interleaved with steps of another operation executed by another process. To formally capture the fact that operations can overlap in this way, we introduce the notion of *histories* as sequences of *events*. *Event*s in the sequential world are invocations or returns of operations. In TSO, they will be an invocation, response or a flush. The sets of events are denoted *Event* and $Event_{TSO}$ respectively. Flushes are performed by the CPU and operate on a particular process's buffer.

[1] We do not need a full formal semantics of TSO, but interested readers are referred to [16, 15] for formal definitions.

A method is *pending* if it has been invoked but has not yet returned. A history is *sequential* if all invoke events are immediately followed by their matching returns. Where this is not the case, methods overlap. A *quiescent* state is one in which there are no pending methods, and all buffers have been flushed.

Invocations and returns of operations from a set I are performed by a particular process from a set P with an input or output value V. We let $\perp$ denote empty inputs/outputs and assume that $\perp \in V$. Thus we define:

$$\begin{aligned} \mathit{Event} &::= \mathit{inv}\langle\langle P \times I \times V\rangle\rangle \mid \mathit{ret}\langle\langle P \times I \times V\rangle\rangle \\ \mathit{Event}_{TSO} &::= \mathit{inv}\langle\langle P \times I \times V\rangle\rangle \mid \mathit{ret}\langle\langle P \times I \times V\rangle\rangle \mid \mathit{flush}\langle\langle P\rangle\rangle \end{aligned}$$

Example 2. A TSO history corresponding to the sequence s_1 above is:[2]

$$\begin{aligned} h_1 = \langle & \mathit{inv}(p, \mathtt{set\_x}, 1), \mathit{ret}(p, \mathtt{set\_x},), \mathit{inv}(p, \mathtt{read\_y},), \mathit{ret}(p, \mathtt{read\_y}, 0), \\ & \mathit{inv}(q, \mathtt{set\_y}, 1), \mathit{ret}(q, \mathtt{set\_y},), \mathit{inv}(q, \mathtt{read\_x},), \mathit{ret}(q, \mathtt{read\_x}, 0), \\ & \mathit{flush}(p), \mathit{flush}(q)\rangle \end{aligned}$$

It is in a quiescent state initially and at its end but not anywhere in between. □

Correctness means that the histories of an implementation should correspond 'in some sense' to those of its abstract specification (in which overlapping operations are not possible). Varying the meaning of 'in some sense' results in different correctness conditions [10]. Of these, *linearizability* has been widely used as the correctness criterion in sequentially consistent architectures. However, issues arise in the context of TSO since the direct application of linearizability to TSO requires the natural abstract specification to be weakened (see the approaches of both [9, 3]). Thus it seems that linearizability might impose too many constraints to be a useful criterion for a weak memory model such as TSO since it requires sequential consistency amongst the system processes [10]. Instead, here we use an alternative (weaker) correctness criterion, *quiescent consistency* [17]. Informally it states that operations separated by quiescent states should appear to take effect in their (real-time) order.

Quiescent consistency has been recently formalised in [5] for sequentially consistent architectures and a proof method developed for it. Our first task therefore is to adapt the definition for a TSO model. First of all some preliminaries. For a history h, $\#h$ is the *length* of the sequence, and $h(n)$ its nth element (for $n : 1..\#h$). Predicates $\mathit{inv}?(e)$, $\mathit{ret}?(e)$, and $\mathit{flush}?(e)$ are used to check whether an event $e \in \mathit{Event} \cup \mathit{Event}_{TSO}$ is an invoke, return or flush, respectively. We let $e.p \in P$, $e.i \in I$ and $e.v \in V$ be the process executing the event e, the operation to which the event belongs, and the input/output value of v, respectively. Furthermore, for a TSO event e that is a return or flush, we assume $e.bv$ is the boolean corresponding to the event, which holds iff all local buffers are empty immediately after e occurs. Finally, we let $\mathit{Ret}!$ be the set of all TSO return events.

We now need a preliminary definition saying what it means to be a matching pair of invocation and return, a pending invocation, a legal history (where each process calls at most one operation at a time), and a quiescent history (a history which is in a quiescent state at its end). Unlike, earlier work, we record concrete flush events in the concrete histories to handle TSO memory, and hence, the definition below differs from [5, 6], which were defined for SC architectures.

[2] We elide empty inputs or outputs in the events, e.g., write $\mathit{ret}(p, \mathtt{set\_x},)$ for $\mathit{ret}(p, \mathtt{set\_x}, \perp)$.

Definition 1. *Suppose $h \in seq\,Event \cup seq\,Event_{TSO}$. Two positions $m, n : 1..\#h$ form a* matching pair *iff $mp(m,n,h)$ holds; n in h is a* pending invocation *iff $pi(n,h)$ holds; h is* legal *iff $legal(h)$ holds; and h is* quiescent *iff $qu?(h)$ holds, where:*

$$\begin{aligned}
mp(m,n,h) \mathrel{\widehat{=}}{} & h(m).p = h(n).p \land h(m).i = h(n).i \land inv?(h(m)) \land ret?(h(n)) \land \\
& \forall k \bullet m < k < n \land h(k).p = h(m).p \Rightarrow flush?(h(k)) \\
pi(n,h) \mathrel{\widehat{=}}{} & inv?(h(n)) \land \forall m \bullet n < m \leq \#h \land h(m).p = h(n).p \Rightarrow flush?(h(n)) \\
legal(h) \mathrel{\widehat{=}}{} & h \in seq\,Event_{TSO} \land h \neq \langle\,\rangle \land inv?(h(1)) \land \\
& \forall n : 1..\#h \bullet \textbf{if } inv?(h(n)) \textbf{ then } pi(n,h) \lor \exists m \bullet mp(n,m,h) \\
& \qquad \textbf{elseif } ret?(h(n)) \textbf{ then } \exists m \bullet mp(m,n,h) \\
qu?(h) \mathrel{\widehat{=}}{} & legal(h) \land h(\#h).bv \land \forall n : 1..\#h \bullet \neg pi(n,h)
\end{aligned} \qquad \square$$

Both linearizability and quiescent consistency are defined by comparing the histories generated by concurrent implementations with the sequential histories of some given abstract atomic specification. Here we will adapt the standard definitions to TSO.

Our formal definitions of linearizability and quiescent consistency for TSO are given below. Both are defined using a function *smap* that maps the indices of the concurrent history to a sequential history, and linearizability uses an additional function *complete* that removes all pending invokes. Bijections from X to Y are denoted $X \rightarrowtail\!\!\!\!\twoheadrightarrow Y$. We assume a function $remflush(h,z)$ which transforms h to z by removing all flushes from h, but keeps the order of all other events in h the same. Function *remflush* can be defined recursively, but its formal definition is elided here for space reasons.

$$\begin{aligned}
eveq(e,e') \mathrel{\widehat{=}}{} & e \in Event \land e' \in Event_{TSO} \land e.p = e'.p \land e.i = e'.i \land e.v = e'.v \\
smap(h,f,hs) \mathrel{\widehat{=}}{} & \exists z : \mathrm{seq}\,Event_{TSO} \bullet remflush(h,z) \land f \in 1..\#z \rightarrowtail\!\!\!\!\twoheadrightarrow 1..\#hs \land \\
& \quad \forall m,n : 1..\#z \bullet eveq(hs(f(n)), z(n)) \land \\
& \qquad (mp(m,n,z) \Rightarrow f(m)+1 = f(n))
\end{aligned}$$

Definition 2 (Linearizability and Quiescent consistency on TSO). *Let h be a TSO history, hs a sequential history. The history h is said to be* quiescent consistent *with hs iff $qcons(h,hs)$ holds and* linearizable *with respect to hs iff $lin(h,hs)$ holds, where:*

$$\begin{aligned}
qcons(h,hs) \mathrel{\widehat{=}}{} & \exists f \bullet smap(h,f,hs) \land \\
& \quad \forall m,n,k : 1..\#h \bullet m < n \land m \leq k \leq n \land \\
& \qquad qu?(h[1..k]) \land ret?(h(m)) \land inv?(h(n)) \Rightarrow f(m) < f(n) \\
lrel(h,hs) \mathrel{\widehat{=}}{} & \exists f \bullet smap(h,f,hs) \land \\
& \quad \forall m,n,m',n' : 1..\#h \bullet \\
& \qquad n < m' \land mp(m,n,h) \land mp(m',n',h) \Rightarrow f(n) < f(m') \\
lin(h,hs) \mathrel{\widehat{=}}{} & \exists h_0 : seq\,Ret! \bullet legal(h \frown h_0) \land lrel(complete(h \frown h_0), hs)
\end{aligned} \qquad \square$$

The key point to note is that quiescent consistency allows the operations of processes *between* quiescent states to be reordered, whereas linearizablity does not. As in [5], we have the following property.

Proposition 1 (Linearizability implies quiescent consistency).
For any $h \in seq\,Event_{TSO}$ and $hs \in seq\,Event$, $lin(h,hs) \Rightarrow qcons(h,hs)$. $\square$

Example 3. Let us return to Example 1 above. In what sense is this correct with respect to an abstract specification which has one operation for each concrete one? Consider h_1

again. Because of the effect of the local buffers, both read operations return 0. This is only possible at the abstract level if the reads occur before the writes. For example, h_1 could be mapped to sequential history:

$$\begin{aligned}h_2 = \langle &\mathit{inv}(p, \texttt{read\_y},), \mathit{ret}(p, \texttt{read\_y}, 0), \mathit{inv}(q, \texttt{read\_x},), \mathit{ret}(q, \texttt{read\_x}, 0),\\ &\mathit{inv}(p, \texttt{set\_x}, 1), \mathit{ret}(p, \texttt{set\_x},), \mathit{inv}(q, \texttt{set\_y}, 1), \mathit{ret}(q, \texttt{set\_y},)\rangle\end{aligned}$$

Such a reordering is possible under quiescent consistency, but not linearizability. □

This example highlights a typical consequence of using a TSO architecture. We should allow for programmers to exploit such consequences in order to improve the efficiency of their algorithms. Therefore, in some circumstances it makes sense to adopt quiescent consistency as the correctness criterion for TSO. The only existing work on correctness on TSO [3, 9] has looked at linearizability, and to do so has needed to modify the high-level requirements of the algorithms by either adding implementation-level details such as buffers and flushes, or nondeterminism reflecting operation reorderings to the abstract specifications. There has been no work, as far as we are aware, on quiescent consistency as the correctness criterion for TSO.

3 Modelling an Algorithm on the TSO Architecture

As a more complex motivating example, we examine the Linux locking mechanism *seqlock* [13], which allows reading of shared variables without locking the global memory, thus supporting fast write access. We begin by showing that while *seqlock* is linearizable on a standard architecture, it is neither linearizable nor quiescent consistent on TSO without the use of memory barriers. We then turn our attention to a one-writer variant of *seqlock* (based on the non-blocking write protocol of [12]) which we show is quiescent consistent in Section 4.

Example 4. In the usual *seqlock* algorithm all processes can read and write. A process wishing to *write* to the shared variables `x1` and `x2` *acquires* a lock and increments a counter *c*. It then proceeds to write to the variables, and finally increments *c* again before *releasing* the lock. The lock ensures synchronisation between writers, and the counter *c* ensures the consistency of values read by other processes. The two increments of *c* ensure that it is odd when a process is writing to the variables, and even otherwise. Hence, when a process wishes to *read* the shared variables, it waits in a loop until *c* is even before reading them. Also, before returning it checks that the value of *c* has not changed (i.e., another write has not begun). If it has changed, the process starts over. A typical implementation of *seqlock* (based on that in [3]) is given in Figure 1. A local variable `c0` is used by the `read` operation to record the (even) value of `c` before the operation begins updating the shared variables. In general, the `release` operation does not include a fence instruction and so does not flush the buffer. □

Abstract specification - AS. The algorithm is captured abstractly in Z [20], a state-based specification formalism that allows specification of data types by defining their state (variables), initial state and operations. All these are given as *schemas*, consisting of variable declarations plus predicates further constraining the variables. Input and

```
word x1 = 0, x2 = 0;
word c = 0;                     read(out word d1,d2) {
                                   word c0;
write(in word d1,d2) {             do {
   acquire;                           do {
   c++;                                  c0 = c;
   x1 = d1;                           } while (c0 % 2 != 0);
   x2 = d2;                           d1 = x1;
   c++;                               d2 = x2;
   release; }                      } while (c != c0); }
```

Fig. 1. Seqlock implementation

output variables are decorated by '?' and '!', respectively, and notation v' denotes the value of a variable v in the post state of an operation. Unprimed variables of a schema S are introduced into another schema by including S in the declaration, and both unprimed and primed variables are introduced using ΞS or ΔS, the former constraining variables to remain unchanged. For the program in Fig. 1, the abstract specification is:

AS

$x_1, x_2 : \mathbb{N}$

ASInit

AS

$x_1 = x_2 = 0$

Write$_q$

ΔAS

$d_1?, d_2? : \mathbb{N}$

$x_1' = d_1? \land x_2' = d_2?$

Read$_q$

ΞAS

$d_1!, d_2! : \mathbb{N}$

$d_1! = x_1 \land d_2! = x_2$

The abstract state (as defined by schema *AS*) consists of two variables x_1 and x_2 of type $\mathbb{N}$, representing `x1` and `x2` in Fig. 1, respectively. The initial state is given by *ASInit*, which ensures that execution begins in a state in which the value of both x_1 and x_2 is 0. Parameterised schemas $Write_q$ and $Read_q$, where q denotes the process performing the step, represent abstractions of the fine-grained operations `write` and `read` in Fig. 1, respectively. $Write_q$ (atomically) sets the values of x_1 and x_2 by ensuring the values of x_1' and x_2' equal to $d_1?$ and $d_2?$, respectively. $Read_q$ sets the outputs $d_1!$ and $d_2!$ to x_1 and x_2, respectively.

Histories of the abstract specification are generated by initialising the state as specified by *ASInit*, then repeatedly choosing a process q and schema $Write_q$ or $Read_q$ non-deterministically, and transitioning to the next state as specified by the chosen schema.

Proposition 2. *Seqlock is linearizable with respect to the abstract specification AS on a sequentially consistent architecture.*

Choosing the final statement of each concrete operation as its *linearization point* (i.e., the point where the operation appears to take effect), the proposition can be proved using the existing approach of Derrick et al. [6]. □

Proposition 3. *Seqlock is not linearizable with respect to the abstract specification AS on the TSO architecture, nor is it quiescent consistent.*

Proof By Proposition 1 linearizability does not hold if quiescent consistency does not hold. To show quiescent consistency does not hold, we provide a counter example, which follows from the fact that flushes from successive writes can interleave resulting in inconsistent reads. For example, in the following concrete history process r reads the values 3 and 2 for `x1` and `x2`, respectively, which cannot occur according to the abstract specification:

$$\langle \mathit{inv}(p, \mathtt{write}, (1,2)), \mathit{ret}(p, \mathtt{write},), \mathit{inv}(q, \mathtt{write}, (3,4)), \mathit{ret}(q, \mathtt{write},),$$
$$\mathit{flush}(p), \mathit{flush}_F(p), \mathit{flush}(q), \mathit{flush}(q), \mathit{flush}(q), \mathit{flush}(q), \mathit{flush}(p), \mathit{flush}(p),$$
$$\mathit{inv}(r, \mathtt{read},), \mathit{ret}(r, \mathtt{read}(3,2))\rangle$$

The four flushes for each of p and q correspond to the flushing of the first write to `c`, the write to `x1`, the write to `x2`, and the second write to `c`, respectively. Note that this counter-example is only possible since the `write` operations may not include a fence. □

To avoid such inconsistent behaviour in practice, a memory barrier is required at the end of the `write` operation. Since reads cannot complete while a `write` operation is pending, with this memory barrier there are no behaviours possible on TSO other than those possible on a sequentially consistent architecture. Hence, the algorithm can be proved linearizable by Proposition 2.

To illustrate correctness proofs on a TSO architecture further, we examine a variant of *seqlock* in which all processes can read, but only one can write. In this case, no writer lock is required and the `write` operation can be simplified by removing the `acquire` and `release` commands. To verify quiescent consistency (or indeed linearizability) we need a formal specification of this system, which we give now.

Concrete specification - CS. Let P be a set of processes and PC denote program counter values.

$$PC ::= 1 \mid w_1 \mid w_2 \mid w_3 \mid r_1 \mid r_2 \mid r_3 \mid r_4$$

The state CS comprises both the global variables, and program counters, local variables and buffers for each process.

$$\begin{array}{|l} \mathit{CS} \\ \hline x_1, x_2, c : \mathbb{N} \\ pc : P \rightarrow PC \\ d_1, d_2, c_0 : P \rightarrow \mathbb{N} \\ b : P \rightarrow \mathit{seq}(\{\mathsf{x}_1, \mathsf{x}_2, \mathsf{c}\} \times \mathbb{N}) \\ \hline \end{array} \qquad \begin{array}{|l} \mathit{CSInit} \\ \hline \mathit{CS} \\ \hline x_1 = x_2 = c = 0 \\ \forall\, q : P \bullet pc(q) = 1 \wedge b(q) = \langle\,\rangle \\ \hline \end{array}$$

The elements of the processes' buffers (denoted by variable b) are ordered pairs, the first element of which identifies a global variable (using a label, e.g., x_1, written in sans serif), and the second the value written to it by the process. To simplify the presentation of the operation specifications, we adopt the following two conventions:

1. Any values (of variables or in the range of functions) that are not explicitly defined in an operation are unchanged.

2. $\overline{x_1}(q)$ denotes the value of x_1 read by a process q. This value is either the most recent in its buffer or, when no such value exists, the value of the global variable x_1. Simarly, for $\overline{x_2}(q)$ and $\overline{c}(q)$.

Let $p : P$ denote the writer process. The `write` operation is captured by four operations in Z: one for each of its lines (given that the `acquire` and `release` commands have been removed). The subscript p acts as a parameter to the operations.

$$\begin{array}{|l}
\multicolumn{1}{l}{W1_p} \\
\Delta CS \\
d_1?, d_2? : \mathbb{N} \\
\hline
pc(p) = 1 \wedge pc'(p) = w_1 \\
d_1'(p) = d_1? \wedge d_2'(p) = d_2? \\
b'(p) = b(p) \frown \langle(\mathsf{c}, \overline{c}(p) + 1)\rangle \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{W2_p} \\
\Delta LS \\
\hline
pc(p) = w_1 \wedge pc'(p) = w_2 \\
b'(p) = b(p) \frown \langle(\mathsf{x}_1, d_1)\rangle \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{W3_p} \\
\Delta CS \\
\hline
pc(p) = w_2 \wedge pc'(p) = w_3 \\
b'(p) = b(p) \frown \langle(\mathsf{x}_2, d_2)\rangle \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{W4_p} \\
\Delta CS \\
\hline
pc(p) = w_3 \wedge pc'(p) = 1 \\
b'(p) = b(p) \frown \langle(\mathsf{c}, \overline{c}(p) + 1)\rangle \\
\hline
\end{array}$$

The `read` operation is captured by 5 operations in Z: $R1_q$ for one iteration of the inner do-loop[3], $R2_q$ for the assignment to local variable `d1`, $R3_q$ for the assignment to local variable `d2`, $R4_q$ for starting a new iteration of the outer do-loop (when $c \neq c_0$), and $R5_q$ for returning the read values (when $c = c_0$). In each case, the subscript $q : P$ is a parameter identifing the process performing the operation. There is also an operation $Flush_p$ corresponding to a CPU flush of the writer process's buffer.

$$\begin{array}{|l}
\multicolumn{1}{l}{R1_q} \\
\Delta CS \\
\hline
pc(q) = 1 \vee pc(q) = r1 \\
c_0'(p) = c \wedge \textbf{if } c_0'(q) \bmod 2 \neq 0 \\
\qquad\qquad\quad \textbf{then } pc'(q) = r1 \\
\qquad\qquad\quad \textbf{else } pc'(q) = r2 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{R2_q} \\
\Delta CS \\
\hline
pc(q) = r_2 \wedge pc'(q) = r_3 \\
d_1'(q) = \overline{x_1}(q) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{R3_q} \\
\Delta CS \\
\hline
pc(q) = r_3 \wedge pc'(q) = r_4 \\
d_2'(q) = \overline{x_2}(q) \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{R4_q} \\
\Delta CS \\
\hline
pc(q) = r4 \wedge c_0(q) \neq \overline{c}(q) \\
pc'(q) = r1 \\
\hline
\end{array}$$

[3] Any command, and in particular a write to `c`, occurring between the assignment to `c0` and the check of the loop condition will have the same effect as if it occurred after the check. Hence, no potential behaviour is prohibited by modelling the two commands by a single operation.

$$\begin{array}{|l}
\multicolumn{1}{l}{R5_q} \\
\Delta CS \\
d_1!, d_2! : \mathbb{N} \\
\hline
pc(q) = r4 \land c_0(q) = \overline{c}(q) \\
d_1! = d_1(p) \land d_2! = d_2(q) \\
pc'(q) = 1
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{Flush_p} \\
\Delta CS \\
\hline
b(p) \neq \langle\,\rangle \\
(b(p)(1)).1 = \mathsf{x}_1 \Rightarrow x_1' = (b(p)(1)).2 \\
(b(p)(1)).1 = \mathsf{x}_2 \Rightarrow x_2' = (b(p)(1)).2 \\
(b(p)(1)).1 = \mathsf{c} \Rightarrow c' = (b(p)(1)).2 \\
b'(p) = tail\, b(p)
\end{array}$$

4 Showing Quiescent Consistency on the TSO Architecture

With this model in place we now consider how we can verify that it is indeed quiescent consistent. First of all we consider why linearizability is not appropriate.

Proposition 4. *Seqlock with one writer process is not linearizable with respect to the abstract specification AS on a TSO architecture.*

This follows from Example 5 below, which gives a history in which reads from the writer process p and another process q occur in an order that is not possible at the abstract level. □

Example 5. The following concrete history is possible.

$$\langle inv(p, \texttt{write}, (1,2)), ret(p, \texttt{write},), inv(p, \texttt{read},), ret(p, \texttt{read}, (1,2)),$$
$$inv(q, \texttt{read},), ret(q, \texttt{read}, (0,0)), flush(p), flush(p), flush(p), flush(p)\rangle$$

The first flush occurs after q's read so `c` will be even, which allows the read to proceed. The first read in the history (by p) reads the values of `x1` and `x2` from p's buffer. The second (by q) reads from the global memory. The overall effect is that the second read returns older values than the first; hence, there is no corresponding abstract history. □

Burckhardt et al. [3] prove this variant of *seqlock* is linearizable on TSO. However, in order to do this, they are forced to use an abstract specification that, like the concrete algorithm, has local buffers and CPU flushes. Hence, reading of older values after newer values have been read (as in the history above) is allowed by the abstract specification. It is our goal, however, to show correctness with respect to the stronger, and more intuitive, abstract specification *AS* — since an abstract description of *seqlock* should not mention buffers and flushes.

Under quiescent consistency, the above history could be reordered as the following abstract sequential history:

$$hs = \langle inv(q, \texttt{read},), ret(q, \texttt{read}, (0,0)), inv(p, \texttt{write}, (1,2)), ret(p, \texttt{write},),$$
$$inv(p, \texttt{read},), ret(p, \texttt{read}, (1,2))\rangle$$

Proposition 5. *Seqlock with one writer process is quiescent consistent with respect to abstract specification AS on a TSO architecture.* □

We describe a proof methodology in Section 4.1 and then give an outline proof of this proposition using the schemas from *AS* and *CS* in Section 4.2.

4.1 Simulation Rules for Quiescent Consistency

We adapt a refinement-based proof method for verifying quiescent consistency on sequentially consistent memory models defined in [5]. Let our abstract specification be given as $A = (AState, AInit, (AOP_{p,i})_{p \in P, i \in I})$ and concrete specification be given as $C = (CState, CInit, (COP_{p,j})_{p \in P, j \in J})$ where the sets I and J are used to index the abstract and concrete operations and P is the set of all process identifiers. The function $abs : J \to I$ maps each concrete operation to the abstract operation it implements. In the definitions below, we treat operations as functions in the following two ways: $AOp_{p,i}(in_i, out_i)$ denotes an operation with its input and output parameters, and in $COp_{p,j}(in, cs, cs')$ we have made the before and after states explicit.

The simulation rules for quiescent consistency use a non-atomic, or coupled, simulation [7] which matches the concrete return events that result in a quiescent state with a sequence of abstract operations, and (abstractly) views all other concrete events as *skips*. For this to work, we need to keep track of the progress of the concrete operations in non-quiescent states. Thus we extend the retrieve relation R (between abstract and concrete states) with a history H, giving a family of retrieve relations R^H. For transitions to a quiescent state, we need to match up with a sequence of *all* abstract operations corresponding to the invoke and return events occurring in H. Quiescent consistency allows us to potentially reorder H to achieve this.

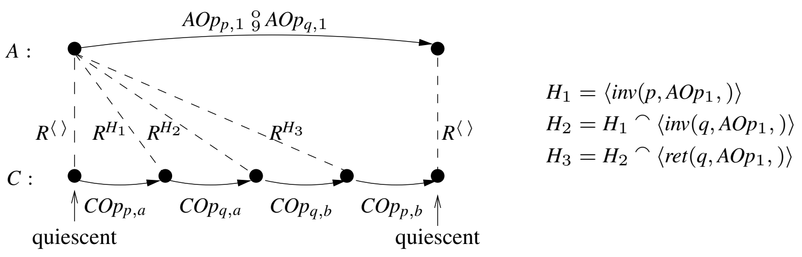
$AOp_{p,1} \mathbin{;} AOp_{q,1}$
A :
C :
$R^{\langle\rangle}$
R^{H_1}
R^{H_2}
R^{H_3}
$R^{\langle\rangle}$
$COp_{p,a}$
$COp_{q,a}$
$COp_{q,b}$
$COp_{p,b}$
quiescent
quiescent
$H_1 = \langle inv(p, AOp_1,) \rangle$
$H_2 = H_1 \frown \langle inv(q, AOp_1,) \rangle$
$H_3 = H_2 \frown \langle ret(q, AOp_1,) \rangle$

Fig. 2. Coupled simulation for some example run

Figure 2, taken from [5], illustrates an example where the abstract operation $AOp_{p,1}$ is implemented as $COp_{p,a} \mathbin{;} COp_{p,b}$ (so *abs* maps both a and b to 1). Processes p and q execute concrete steps. In the initial and final quiescent states, the systems are related by $R^{\langle\rangle}$. In non-quiescent states the systems are related by a retrieve relation that has recorded (via invocation and return events) the concrete operations that have completed. These will ultimately have to be matched when transitioning to a quiescent state. As with all simulations, to apply the proof method one needs to define the simulation rules, and prove that all the squares (and triangles) in diagrams such as Figure 2 commute.

Notation: The definition of coupled simulation uses predicates $inv?(Op)$, $ret?(Op)$, $int?(Op)$, and $flush?(Op)$ that hold iff the concrete operation Op is an invocation, return, internal (i.e., neither invoke, return or flush), and flush event respectively.

To allow the concrete reordering we write $hs \simeq H$ for two histories $hs : \mathrm{seq}\, Event$ and $H : \mathrm{seq}\, Event_{TSO}$ iff $hs \sim H \setminus \{\!|\, flush \,|\!\}$ where $\sim$ is permutation equivalence and $H \setminus \{\!|\, flush \,|\!\}$ removes all flushes from the history H.

Furthermore, we let *AOP* denote the set of all abstract operations and define a function *hist* which constructs the sequential history corresponding to a sequence of abstract operations:

$$hist(\langle AOp_{p_1,1}(in_1, out_1), \dots, AOp_{p_n,n}(in_n, out_n)\rangle) \mathrel{\widehat=} \\ \langle inv(p_1, AOp_1, in_1), ret(p_1, AOp_1, out_1), \dots, inv(p_n, AOp_n, in_n), ret(p_n, AOp_n, out_n)\rangle$$

For a sequential history *hs* and abstract states *as* and *as*$'$ we define

$$seqhist(hs, as, as') \mathrel{\widehat=} \exists\, aops : AOP^* \bullet aops(as, as') \wedge hist(aops) = hs,$$

which holds iff (a) there is some abstract sequence of operations *aops* whose composition (in order) is a relation between *as* and *as*$'$, and (b) the sequential history generated from *aops* is *hs*.

Example 6. The abstract history *hs* of Example 5 maps the state $as = \{x_1 \mapsto 0, x_2 \mapsto 0\}$ to $as' = \{x_1 \mapsto 1, x_2 \mapsto 2\}$. Hence, we have $seqhist(hs, as, as')$. □

A coupled simulation for proving quiescent consistency is given below. It uses the following definition where *h* is an abstract history and *cs* is a concrete state:

$$qu(h, cs) \mathrel{\widehat=} \forall\, n : 1..\#h \bullet \neg pi(n, h) \wedge \forall\, p : P \bullet cs(b(p)) = \langle\,\rangle$$

which holds iff (a) there are no pending invocations in *h*, and (b) all process buffers are empty in *cs*, where we assume *b* models the buffer for each process. This is used to determine when an execution is in a quiescent state. For example, in seqlock `read`s are completed by their returns, whereas `write`s require at least one flush after their return to complete. Therefore, we reach a quiescent state in one of the following two ways. Either all buffers are empty and the lasting pending `read` returns, or all processes are idle and a flush empties the last non-empty buffer.
Note that unlike Definition 2, which uses concrete histories to determine quiescence, Definition 3 below uses both the histories *H* and the concrete state built up in R^H.

Definition 3 (QC Coupled simulation for TSO). *Let A and C be abstract and concrete specifications. Let* $H : seq\,Event_{TSO}$. *A family of relations* $R^H \subseteq AState \times CState$ *is a* QC (quiescent consistent) coupled simulation *from A to C iff* $\forall\, as : AState,\ cs, cs' : CState,\ in, out : V,\ p : P,\ i : I,\ j : J$ *such that* $R^H(as, cs)$*, each of the following holds:*

1. $COp_{p,j}(in, cs, cs') \wedge inv?(COp_{p,j}) \Rightarrow R^{H \frown \langle inv(p, AOp_{abs(j)}, in)\rangle}(as, cs')$
2. $COp_{p,j}(cs, cs') \wedge int?(COp_{p,j}) \Rightarrow R^{H}(as, cs')$
3. $COp_{p,j}(cs, cs', out) \wedge ret?(COp_{p,j}) \Rightarrow$

   if $\neg qu(H \frown \langle ret(p, AOp_{abs(j)}, out)\rangle, cs')$ **then** $R^{H \frown \langle ret(p, AOp_{abs(j)}, out)\rangle}(as, cs')$
   else $\exists\, as' : AState \bullet R^{\langle\,\rangle}(as', cs') \wedge$
   $\exists\, hs \bullet hs \simeq H \frown \langle ret(p, AOp_{abs(j)}, out)\rangle \wedge seqhist(hs, as, as')$
4. $COp_{p,j}(cs, cs') \wedge flush?(COp_{p,j}) \Rightarrow$

   if $\neg qu(H \frown \langle flush(p)\rangle, cs')$ **then** $R^{H \frown \langle flush(p)\rangle}(as, cs')$
   else $\exists\, as' : AState \bullet R^{\langle\,\rangle}(as', cs') \wedge$
   $\exists\, hs \bullet hs \simeq H \frown \langle flush(p)\rangle \wedge seqhist(hs, as, as')$

together with the **initialisation** *condition:* $\forall\, ci : CInit \bullet \exists\, ai : AInit \bullet R^{\langle\,\rangle}(ai, ci)$. □

We now describe these proof obligations. Rule 1 is for a step that starts (invokes) an operation, e.g., $W1_p$, where the event corresponding to the step is added to the history H collected thus far. The proof obligation requires that if R^H holds before the invocation then $R^{H \frown \langle inv(p, AOp_{abs(j)}, in)\rangle}$ holds after, and the abstract system does not take a step. Rule 2 applies to steps that are neither invocations nor returns, e.g., $W2_p$, and requires that R^H is maintained. Again, the abstract system does not take a step.

Rule 3 applies to a return step, e.g., $W4_p$, and has two cases. The case is determined by appending the corresponding return event to H and testing whether or not qu holds. For a return to non-quiescent step, the abstract state does not change, but $R^{H \frown \langle ret(p, AOp_{abs(j)}, out)\rangle}$ must hold in the poststate provided R^H holds in the prestate. Return to quiescence is more complicated, and requires that there exists a sequential history hs that is a permutation of $H \frown \langle ret(p, AOp_{abs(j)}, out)\rangle$ such that overall effect of the steps corresponding to hs is equivalent to a transition from the abstracting start state as to as'. Furthermore, as' is related to the concrete poststate cs' via $R^{\langle\rangle}$, where the history collected thus far is empty because the system has returned to quiescence.

These rules are in essence the same as those for sequentially consistent architectures in [5]. For TSO, we need an additional rule for flushes, Rule 4. This rule is similar to Rule 3. For a return to non-quiescent step, the abstract state does not change, and $R^{H \frown \langle flush(p)\rangle}$ must hold in the poststate provided R^H holds in the prestate. Return to quiescence requires that there exists a sequential history hs that is a permutation of $H \frown \langle flush(p)\rangle$ such that overall effect of the steps corresponding to hs is equivalent to a transition from the abstracting start state as to as'. Again, as' is related to the concrete poststate cs' via $R^{\langle\rangle}$.

Quiescent consistency, as with linearizability, is a *safety property* and no liveness is guaranteed. Therefore, Definition 3 does not mention any applicability conditions.

Following the proof strategy in [5], it can be shown that coupled simulation is a sound proof technique for quiescent consistency (the proof of this follows from the definition).

Theorem 1. *Let A and C be abstract and concrete specifications, respectively. If there is a coupled simulation R^H from A to C, then C is quiescent consistent wrt. A.* $\square$

4.2 Proof Outline for Seqlock

To apply Definition 3 one needs to define R^H and also give the explicit reordering of the concrete history on returning to quiescence (conditions 3 and 4). R^H is a conjunction of a number of individual cases corresponding to possible values of the buffer in any state. The proof consists of a number of small proof steps which individually are not complicated but we do not have space to give them all here. Rather we just aim to give a flavour of what is involved.

First we need to determine which condition(s) of Definition 3 needs to be proved for each of the concrete Z operations. Condition 1 needs to be proved for $W1_p$ and $R1_q$ which are the invocation events of the `write` and `read` operations respectively. Condition 3 needs to be proved for $W4_p$ and $R5_q$ which are the return events of the `write` and `read` operation respectively, and condition 4 for the occurrence of $Flush_p$. Condition 2 needs to be proved for all other operations.

1. Defining $R^{\langle\rangle}$: $R^{\langle\rangle}$ relates abstract states and quiescent concrete states. The latter are those in which $pc(q) = 1$ for all processes q, and the buffer of the writer process p is empty. In these states, the abstract and concrete values of x_1 and x_2 are equal, and c is even. That is, letting $A.x_1$ and $A.x_2$ denote the abstract variables x_1 and x_2, $R^{\langle\rangle}$ is true when

$$A.x_1 = x_1 \wedge A.x_2 = x_2 \wedge c \bmod 2 = 0 \wedge (\forall q : P \bullet pc(q) = 1) \wedge b(p) = \langle\rangle$$

2. Defining R^H: For $H \neq \langle\rangle$, R^H includes a number of conjuncts depending on the values of pc and b for the individual processes. For example, when H's last event is an invocation of the `write` operation with input values $d1?$ and $d2?$, R^H includes the following conjuncts.

– If $pc(p) = w_1$ the inputs from the pending `write` operation are in $d_1(p)$ and $d_2(p)$.

$$pc(p) = w_1 \Rightarrow d_1(p) = d_1? \wedge d_2(p) = d_2? \tag{1}$$

– If $pc(p) = w_2$ the inputs from the pending `write` operation are either in the last entry of $b(p)$ and $d_2(p)$, or x_1 and $d_2(p)$ when the writer process's buffer has been completely flushed.

$$pc(p) = w_2 \wedge b(p) \neq \langle\rangle \Rightarrow last\, b(p) = (\mathsf{x}_1, d_1?) \wedge d_2(p) = d_2? \tag{2}$$
$$pc(p) = w_2 \wedge b(p) = \langle\rangle \Rightarrow x_1 = d_1? \wedge d_2(p) = d_2? \tag{3}$$

3. Proof obligations for initialisation and non-quiescent states. Given the complete definition of R^H it is possible to prove the initialisation condition and the coupled simulation conditions for each concrete Z operation that does not result in a quiescent state. For example, consider just the conjuncts (1) to (3) above.

The invocation event for the `write` operation is $W1_p$. This operation sets $d1(p)$ to $d_1?$, and $d2(p)$ to $d_2?$, and so establishes the consequent of (1). Since $pc(p) = w1$ in its poststate, (1) to (3) hold.

Operation $W2_p$ is an internal event. It adds $(\mathsf{x}_1, d1(p))$ to the end of $b(p)$. Since (1) holds in the prestate of the operation, $last\, b(p) = (\mathsf{x}_1, d_1?)$ in the poststate. Also since (1) holds in the prestate and the operation does not change $d_2(p)$, $d_2(p) = d_2?$ in the poststate. Hence, the consequent of (2) is established. Since $W2_p$ also establishes $pc(p) = w2$ and $b(p) \neq \langle\rangle$ in its poststate, (1) to (3) hold.

When $pc(p) = w1$ or $pc(p) = w_2$, a $Flush_p$ operation can also result in a non-quiescent state. It does not change $pc(p)$. When $pc(p) = w1$, since the consequent of (1) holds in the prestate of the operation, it will also hold in the poststate since $Flush_p$ does not change $d_1(p)$ or $d_2(p)$. Hence, (1) to (3) hold.

When $pc(p) = w_2$, since $b(p) \neq \langle\rangle$ in the prestate of $Flush_p$ the consequent of (2) holds. If in the poststate $b(p) \neq \langle\rangle$ then, since $Flush_p$ does not change $last\, b(p)$ or $d_2(p)$, the consequent of (2) continues to hold as required. If in the poststate $b(p) = \langle\rangle$ then in the prestate there was only one entry in the buffer which we know from the consequent of (2) is $(\mathsf{x}_1, d_1?)$. Hence, in the operation's poststate we have $x_1 = d_1?$

and, since $Flush_p$ does not change $d_2(p)$, $d_2(p) = d_2?$. Hence, the consequent of (3) holds as required. Therefore in both cases, (1) to (3) hold.

Finally, when $pc(p) = w_1$ or $pc(p) = w_2$, a process other than p can do any of the concrete Z operations capturing the `read` operation, as an invocation or internal event. In each case, since no local variables of p nor any global variables are changed, (1) to (3) will continue to hold.

In the full proof, the above reasoning would be extended to all conjuncts which comprise R^H for each concrete history H beginning from a quiescent state.

4. Proof obligations for quiescent states. The remaining steps of the proof require showing that each concrete Z operation that results in a quiescent state simulates an abstract history which is a reordering of the concrete history since the last quiescent state. As discussed earlier there are two ways of a reaching a quiescent state. The first is when all buffers are empty and the lasting pending `read` returns. In this case, the else condition of Rule 3 applies. The other case is when all processes are idle and a flush empties the last non-empty buffer. In this case the else condition of Rule 4 applies.

To prove the rules we are required to find a reordering of the sequence of the concrete history, which can be determined for both Rule 3 and 4 as follows.

Case 1 : $ret(p, \texttt{read}, (\_,\_))$ occurs between $ret(p, \texttt{write}, (\_,\_))$ and the final $flush(p)$ of that `write`.
In this case, there is no to reorder the operations (since the `read` is from p's buffer and so is consistent with the `write`) and the abstract history corresponds to the order of returns.

Case 2 : $ret(q, \texttt{read}, (\_,\_))$ occurs between $ret(p, \texttt{write}, (\_,\_))$ and the final $flush(p)$ of that `write`.
In this case, the clue for finding a valid reordering is found in Example 5 where a process reads an older value after a newer value has been read. To avoid this situation, we can reorder the concrete history as follows. In the reordered abstract history, we want the `read` by q to occur before the `write` by p. Therefore, we move the return (and if necessary, invocation) of the `read` to be immediately before the return of the `write`. As in Case 1, the order of the abstract history is then the order of the returns. If there is more than one such `read` operation, the order they appear in before the `write` operation is arbitrary.

Case 3 : $ret(p, \texttt{read}, (\_,\_))$ occurs after both $ret(p, \texttt{write}, (\_,\_))$ and the final $flush(p)$ of that `write`.
In this case, there is no to reorder the operations and the abstract history corresponds to the order of returns.

Case 4 : $ret(q, \texttt{read}, (\_,\_))$ occurs after both $ret(p, \texttt{write}, (\_,\_))$ and the final $flush(p)$ of that `write`.
In this case, there is no to reorder the operations and the abstract history corresponds to the order of returns.

The reordered concrete history will have no `read` operations on q while a `write` operation on p is pending or has not yet been completely flushed to the global memory. Hence, there will be no effects from writes being delayed: all reads by processes other than the writer process will occur either before the write begins, or after it has been completely flushed to memory. Therefore, there will be an abstract history corresponding to the reordered concrete one. As an example, consider the following concrete history with a single `write` and three `read`s.

$\langle inv(p, \texttt{write}, (1,2)), inv(r, \texttt{read},), ret(p, \texttt{write},), inv(p, \texttt{read},),\ inv(q, \texttt{read},),$ $ret(q, \texttt{read}, (0,0)), flush(p), flush(p), ret(p, \texttt{read}, (1,2)), flush(p), flush(p),$ $ret(r, \texttt{read}, \texttt{(1,2)})\rangle$

At the end of this history, we are in a quiescent state. All buffers are empty and the lasting pending `read` returns, hence Rule 3 applies. To reorder this, we note that Case 1 applies to the `read` by *p* and Case 4 to the `read` by *r*. Therefore, no reordering is required. For the `read` by *q* Case 2 applies. Therefore, the return of this `read` is moved to immediately before the return of the `write`. In this case, we also need to move the invocation of the `read` (since it occurs after the return of the `write`.

The reordered concrete history is therefore as follows.

$\langle inv(p, \texttt{write}, (1,2)), inv(r, \texttt{read},), inv(q, \texttt{read},), ret(q, \texttt{read}, (0,0)),$ $ret(p, \texttt{write},), inv(p, \texttt{read},), flush(p), flush(p), ret(p, \texttt{read}, (1,2)), flush(p),$ $flush(p), ret(r, \texttt{read}, \texttt{(1,2)})\rangle$

The order of the operations in the corresponding sequential abstract history *hs* is given by the order of returns above:

$\langle inv(q, \texttt{read},), ret(q, \texttt{read}, (0,0)), inv(p, \texttt{write}, (1,2)), ret(p, \texttt{write},),$ $inv(p, \texttt{read},), ret(p, \texttt{read}, (1,2)), inv(r, \texttt{read},), ret(r, \texttt{read}, \texttt{(1,2)})\rangle$.

5 Conclusions

This paper has investigated methods for proving correctness of concurrent algorithms on TSO architectures. Due to the apparent reorderings of reads and writes in a TSO memory model, we have focussed on quiescent consistency as a correctness criterion. We have shown how to model an algorithm and prove quiescent consistency using a simulation-based approach. This was illustrated with a running example based on seqlock, but is applicable to other algorithms running on TSO.

Other work on correctness of algorithms on TSO have altered the definition of linearizability. For example, TSO-TSO linearizability [3] and TSO-SC linearizability [9] have been defined. These approaches, however, prove correctness with respect to abstract specifications which have been altered to include either low-level details of local buffers and CPU flushes (TSO-TSO linearizability), or nondeterminism to account for possible operation reorderings (TSO-SC linearizability). Gotsman *et al.* [9] provide the following abstract specification of seqlock, where the abstract state is modelled as queue: writes are added to the head of the queue, and reads do not return the last value in the queue but any previously written values. In their notation, this is written as follows [9, pg20-21].

```
Queue q = {(0, 0)};              read(out word d1, d2) {
                                   while (*)
write(in word d1, d2) {              q.dequeue();
  q.enqueue(d1,d2); }              (d1,d2) = q.top(); }
```

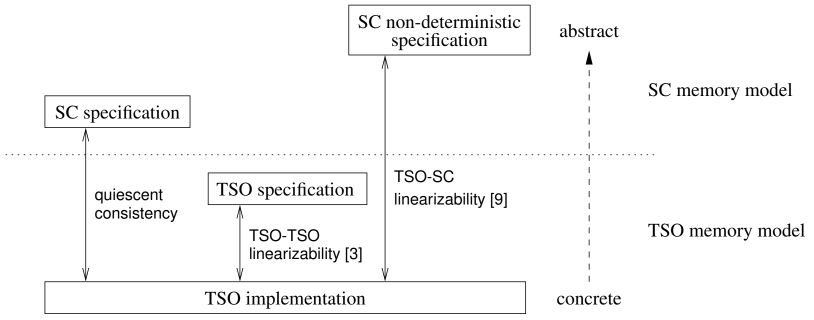

Fig. 3. Comparison of different approaches

This is in contrast to our more natural specification where reads return the most recently written values.

Burckhardt *et al.* also use a more natural specification:

```
word x1 = 0, x2 = 0;

write(in word d1, d2) { lock; x1 = d1; x2 = d2; unlock; }

read(out word d1, d2) { lock; d1 = x1; d2 = x2; unlock; }
```

However, to cope with the effects of TSO memory, each `write` operation of the abstract specification takes place in two atomic steps: a write to a store buffer and a memory flush. Therefore, while the abstract specification is seemingly natural, its underlying semantics is architecturally dependant and includes local store buffers and CPU flushes. As acknowledged by Burckhardt *et al.*, their notion of linearizability is “different from the classical definition of linearizability on a sequentially consistent memory model, which requires methods in the specification to be implemented by one atomic action” [3, pg100].

An overview of our approach in comparison to TSO-TSO and TSO-SC linearizability is given in Fig. 3. TSO-TSO linearizability fails to cross the boundary from the TSO implementation to a sequentially consistent (SC) abstraction, while TSO-SC linearizability crosses this boundary at the cost of a weaker non-deterministic specification. On the other hand, by weakening the linearizability criterion to quiescent consistency, it is possible to prove a relationship with respect to a more intuitive deterministic abstract specification.

References

1. 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.
2. J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *J. ACM*, 41(5):1020–1048, September 1994.

3. 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.
4. 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.
5. J. Derrick, B. Dongol, G. Schellhorn, B. Tofan, O. Travkin, and H. Wehrheim. Quiescent consistency: Defining and verifying relaxed linearizability. In *FM 2014*. Springer, 2014.
6. J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.*, 33(1):4, 2011.
7. J. Derrick and H. Wehrheim. Using coupled simulations in non-atomic refinement. In *ZB 2003*, volume 2651 of *LNCS*, pages 127–147. Springer, 2003.
8. 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.
9. 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. Extended edition http://software.imdea.org/ gotsman.
10. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
11. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
12. H. Kopetz and J. Reisinger. The non-blocking write protocol NBW: A solution to a real-time synchronization problem. In *Real-Time Systems Symposium*, pages 131–137, 1993.
13. C. Lameter. Effective synchronisation on Linux/NUMA systems. In *Gelato Conference*. Silicon Graphics, Inc., 2005.
14. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.
15. S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In T. D'Hondt, editor, *ECOOP 2010*, volume 6183 of *LNCS*, pages 478–503. Springer, 2010.
16. 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.
17. N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011.
18. N. Shavit and A. Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, November 1996.
19. 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.
20. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992.
21. V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.