Proceedings Paper:
Derrick, J. and Smith, G. (2017) An observational approach to defining linearizability on weak memory models. In: Bouajjani, A. and Silva , A., (eds.) Formal Techniques for Distributed Objects, Components, and Systems. FORTE 2017. 37th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems, 19-22 Jun 2017, Neuchâtel, Switzerland. Lecture Notes in Computer Science (10321). Springer, Cham , pp. 108-123. ISBN 9783319602240

https://doi.org/10.1007/978-3-319-60225-7_8

An observational approach to defining linearizability on weak memory models

John Derrick[1] and Graeme Smith[2]

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

Abstract. In this paper we present a framework for defining linearizability on weak memory models. The purpose of the framework is to be able to define the correctness of concurrent algorithms in a uniform way across a variety of memory models. To do so linearizability is defined within the framework in terms of *memory order* as opposed to *program order*. Such a generalisation of the original definition of linearizability enables it to be applied to non-sequentially consistent architectures. It also allows the definition to be given in terms of observable effects rather than being dependent on an understanding of the weak memory model architecture. We illustrate the framework on the TSO (Total Store Order) weak memory model, and show that it respects existing definitions of linearizability on TSO.

1 Introduction

The use of *weak* (or *relaxed*) memory models is standard practice in modern multiprocessor hardware [18]. There are numerous examples including the TSO (Total Store Order) memory model [16, 18], and the memory models of the Power and ARM architectures [1]. TSO is implemented by the x86 architecture used by the chip manufacturers Intel and AMD. The Power architecture is used by IBM, and ARM is the most widely used architecture in mobile devices [10].

All of these architectures provide efficiency gains by reducing the number of accesses to shared memory. For example, in the TSO architecture a buffer is used to store any writes to variables until they can be *flushed* to memory at a convenient time. This time is determined by the hardware to increase efficiency, however if necessary *fences* can be used in the code to force a write to memory. Such instructions flush the entire contents of the buffer to memory.

There is a trade-off between efficiency of the underlying architecture and the use of fences. Furthermore, the presence of both a complicated underlying architecture and associated flushes and fences means there is increased subtlety of the correctness of any given algorithm. This has motivated an increasing interest in verifying the correctness of concurrent algorithms on weak memory models; for example, see [3, 11, 8, 17, 19, 20] for work on TSO.

The standard notion of correctness for concurrent objects is *linearizability* [12]. Given a specification and a concurrent implementation, the idea of linearizability is that any concurrent execution of the implementation must be consistent

with some sequential execution of the specification. The sequential execution is obtained by identifying *linearization points* at which the potentially overlapping concurrent operations are deemed to take effect instantaneously.

A number of approaches have been developed for proving linearizability on *sequentially consistent* architectures, i.e., those without a weak memory model, along with associated tool support [2, 4, 21, 9, 5, 6, 15]. In particular, Derrick et al. [5, 6, 15] have developed a method for proving linearizability supported by the interactive theorem prover KIV [14]. The method consists of proving a number of simulation rules relating a model (a state-transition system) derived from the code and a model representing the abstract specification. This method has been proved sound and complete, the soundness and completeness proofs themselves being done in KIV.

Our recent work, [8], extends the method of Derrick et al. to TSO. To do this, it explicitly adds details of the TSO architecture, i.e., buffers, flushes and fences, into the model derived from the code. To relate this model to the abstract specification, a new set of simulation rules is required to deal with these architectural details which do not occur in the abstract specification.[1] These rules correspond to a new definition of linearizability, referred to in [8] as TSO-linearizability, for which new tool support is required. Due to the explicit modelling of the TSO architecture, we refer to this approach as an *architectural* approach. In [7], we extend this to provide a framework for developing architectural approaches to verification for other weak memory models.

In this paper, we define a new framework for correctness in terms of the *observable* behaviour of weak memory models (as opposed to their underlying architectures). This is beneficial for two reasons. Firstly, the details of many commercial architectures, including Power and ARM, are not available publicly. However, their observable behaviour can be derived via testing: a substantial effort in this direction has already been undertaken for Power and ARM [13]. Secondly, by abstracting from the details of the underlying architecture, the observational approach allows us to use the existing simulation rules and tool support for linearizability. Specifically, the framework does not include architectural details in the model derived from the code and hence, in contrast to the architectural approach, does not require new simulation rules or tool support for each memory model.

The paper is structured as follows. Section 2 introduces the notion of linearizability. Section 3 introduces the TSO memory model and the architectural approach to linearizability on TSO from [8]. Section 4 provides our observational definition of linearizability and shows it is consistent with the architectural definition for TSO. Section 5 discusses the generalisation of the framework to other weak memory models including ARM and Power.

The following is used as a running example throughout the paper.

[1] The related work of Burckhardt [3] and Gotsman [11] avoid this issue by modifiying the abstract specification. We are motivated, however, to allow implementations on TSO to be proved correct with respect to standard specifications of concurrent objects.

1.1 Example: seqlock

The Linux reader-writer mechanism seqlock allows the reading of shared variables without locking the global memory, thus supporting fast write access. It works as follows. A thread wishing to *write* to the shared variables, `x1` and `x2`, say, acquires a software 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 threads as follows. The two increments of `c` ensure that it is odd when a thread is writing to the variables, and even otherwise. Hence, when a thread 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 thread starts over.

Figure 1 provides an abstract specification, in which operations are regarded as atomic. A valid behaviour[2] of seqlock on a sequentially consistent architecture is: $\langle write(t_1, 1, 2);\ read(t_1, 1, 2);\ read(t_2, 1, 2)\rangle$, where, for example, $write(t_1, 1, 2)$ denotes thread t_1 calling the *write* operation with parameters 1 and 2.

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

atomic write(in word d1,d2) {
   x1 = d1;
   x2 = d2;
}
```

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

Fig. 1. seqlock specification

The assumption of atomicity is dropped in the concurrent implementation given in Figure 2 where the statements of operations may be interleaved. Here a local variable `c0` is used by the `read` operation to record the (even) value of `c` before the operation begins updating local variables `d1` and `d2`.

2 Linearizability

Linearizability [12] is the standard correctness criterion for verifying concurrent implementations such as seqlock. Linearizability provides the illusion that each operation executed by a thread takes effect instantaneously at some point between its invocation and its return; this point is known as the *linearization point*. For example, in seqlock the linearization point of the `write` operation is the second store to `c`; after this the values written by the operation can be read by other threads.

Linearizability is defined on *histories*, which are sequences of *events* that can be invocations or returns of operations from a set I and performed by a particular

[2] We use the term *behaviour* to informally refer to a sequence of operations an object may undergo. Later we formalise this as *histories* used in the standard definition of linearizability, and as *executions* used in our observational definition of linearizability.

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

write(in word d1,d2) {
   acquire;
   c++;
   x1 = d1;
   x2 = d2;
   c++;
   release;
}
```

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

Fig. 2. seqlock implementation [3]

thread from a set T. Invocations have an input from domain In and returns have an output from domain Out; both domains contain the value $\perp$ indicating no input or output. On a sequentially consistent architecture we define events and histories as follows:

$$\begin{array}{l} \mathit{Event} \;\widehat{=}\; \mathit{inv}\langle\!\langle T \times I \times \mathit{In} \rangle\!\rangle \mid \mathit{ret}\langle\!\langle T \times I \times \mathit{Out} \rangle\!\rangle \\ \mathit{History} \;\widehat{=}\; \operatorname{seq} \mathit{Event} \end{array}$$

Following [12], each event in a history can be uniquely identified by its operation which we assume is annotated with a subscript representing the occurrence of that operation in the history, so $\texttt{write}_\texttt{n}$ is the nth `write` operation.

Since operations are atomic in an abstract specification, its histories are *sequential*, i.e., each operation invocation will be followed immediately by its return. The histories of a concurrent implementation, however, may have overlapping operations and hence have the invocations and returns of operations separated. However to be *legal*, a history should not have returns for which there has not been an invocation.

Example 1. The following is a possible history of the seqlock implementation:

$$\langle \mathit{inv}(t_1, \texttt{write}_1, (1,2)), \mathit{inv}(t_2, \texttt{read}_1, \perp), \mathit{ret}(t_1, \texttt{write}_1, \perp), \mathit{ret}(t_2, \texttt{read}_1, (1,2)) \rangle$$

Since $\texttt{write}_1$ and $\texttt{read}_1$ overlap, h is not sequential. It is however legal. □

The histories of specifications are also *complete*, i.e., they have a return for each invocation. This is not necessarily the case for implementation histories. To make an implementation history complete, it is necessary to add additional returns for those operations which have been invoked and are deemed to have occurred, and to remove the remaining invocations without matching returns.

Definition 1 (Linearizability [12]). *An implementation of a concurrent object is linearizable with respect to a specification of the object iff for each history h of the implementation, (1) there is a (sequential) history hs of the specification such that the operations of a legal completion of h are identical to those of hs, and (2) the* precedence ordering *of h is preserved by that of hs, i.e., only overlapping operations of h may be reordered with respect to each other in hs.* □

3 The TSO memory model

A weak memory model gives rise to additional behaviours that are not possible on a sequentially consistent architecture. As an example of a weak memory model, we consider the TSO architecture [16, 18].

In TSO, each processor core uses a *store buffer*, which is a FIFO queue that holds pending *stores* (i.e., writes) to memory. When a thread running on a processor core needs to store to a memory location, it enqueues the store to the buffer and continues computation without waiting for the store to be committed to memory. Pending stores do not become visible to threads on other cores until the buffer is *flushed*, which commits (some or all) pending stores to memory. The value of a memory location *loaded* (i.e., read) by a thread is the most recent in that processor's local buffer, and only from the memory if there is no such value in the buffer (i.e., initially or when all stores for that location have been flushed). The use of local buffers allows a load by one thread, occurring after a store by another, to return an older value as if it occurred before the store.

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

On TSO for example, when we run seqlock the `acquire` command of the software lock necessarily has a fence to ensure synchronization between writer threads, however a fence is not required by the `release` command, the effect of which may be delayed. This can lead to unexpected behaviour on TSO. For example, $\langle write(t_1, 1, 2);\ read(t_1, 1, 2);\ read(t_2, 0, 0)\rangle$ is a possible behaviour if t_1's local buffer is not flushed until after t_2's *read*.

The effects of weak memory models, such as TSO, can be understood in terms of potential reordering of program *commands*, i.e., atomic interactions with memory such as loads, stores and fences. The order that the commands of a program p occur in code is captured by the *program order*, which we denote by $<_p$. On a sequentially consistent architecture each thread preserves the program order. However, this is not the case on weak memory models, including TSO. In particular, the order of a *store* occurring before a *load* in TSO is not preserved in the shared memory unless the store is flushed before the load occurs. To formalise such effects we introduce a *memory order*, which we denote by $<_{m(p)}$, and which denotes the order the commands of program p take effect in the shared memory. The effect of TSO can then be characterised by saying that $load <_p store \Rightarrow load <_{m(p)} store$, etc., but that $store <_p load$ does not imply $store <_{m(p)} load$.

These effects are summarised in the following table taken from [18]. The commands include an atomic *read-modify-write*, RMW (e.g., a compare-and-swap (CAS)), and a fence. To be atomic, the former needs to write to memory immediately and hence necessarily includes a fence on TSO (since the write will be placed at the end of the FIFO store buffer). In the table, X denotes an enforced ordering and B denotes that commands can be reordered but *bypassing* is required if the commands are to the same variable. Bypassing means that the

value read is the one that was most recently written by the thread even if it is not yet in the shared memory.

TSO		Command 2			
		load	store	RMW	fence
Command 1	load	X	X	X	X
	store	B	X	X	X
	RMW	X	X	X	X
	fence	X	X	X	X

3.1 Linearizability on TSO

The effect of store buffers means that it is necessary to adapt the linearizability definition for TSO. This is done in [8, 7] by considering how the histories of the implementation are altered, and defining a transformation which then allows concurrent histories to be compared with abstract ones.

To do this, the flush commands are recorded as special events in the TSO histories. Such an event is identified by the thread from whose buffer a value is flushed, and either an operation, if the flush is of the last value written by the operation, or $\bot$ otherwise. Events and histories on TSO are then defined as follows:

$$Event_{TSO} \mathrel{\widehat{=}} inv\langle\langle T \times I \times In\rangle\rangle \mid ret\langle\langle T \times I \times Out\rangle\rangle \mid flush\langle\langle T \times (I \cup \{\bot\})\rangle\rangle$$
$$History_{TSO} \mathrel{\widehat{=}} \operatorname{seq} Event_{TSO}$$

The predicate $flush?(e)$ holds for an event $e \in Event_{TSO}$ iff e is a flush event.

In a sequentially consistent architecture an operation by a thread takes effect at some point between its invocation and return. On a weak memory model, however, the effect of an operation may be delayed until some, or all, of its stores have been flushed. On TSO an operation may actually take effect at any time up to the flush of the last value written by the operation. Implementation histories are thus transformed to reflect this by extending the duration of operations which perform stores: the *effective return* of an operation in a TSO history is either the flush of the final value written by the operation or the return of the operation, whichever occurs later in the history.[3] To represent this, a transformation is defined on histories by:

- moving the return of an operation to replace the final flush for the operation when such a flush occurs after the return, and
- removing all other flushes.

This is encapsulated in the following definition, where for a sequence s, $head\ s$ is the first element of s, $tail\ s$ is s without the first element, $s \oplus \{n \mapsto v\}$ replaces

[3] This principle is also used in other work on linearizability in TSO [19].

the nth value of s with value v, $\#s$ is the length of s, $s(n)$ is the nth element of s, and $s \frown t$ is the concatention of s with a sequence t:

$$trans(h) \;\widehat{=}\; \begin{cases} \langle\,\rangle & \text{if } h = \langle\,\rangle \\ trans(tail\, h) & \text{if } \mathit{flush}?(head\, h) \\ trans(tail(h \oplus \{n \mapsto head\, h\})) & \text{if } DelayedRet(h, n),\ n \leq \#h \\ \langle head\, h\rangle \frown trans(tail\, h) & \text{otherwise} \end{cases}$$

where $DelayedRet(h, n) \;\widehat{=}\; ret?(head\, h) \wedge \mathit{flush}?(h(n)) \wedge (head\, h).i = h(n).i$.

Example 2. One history h_{TSO} of the behaviour $\langle write(t_1, 1, 2);\ read(t_1, 1, 2);\ read(t_2, 0, 0)\rangle$ is:

$$\begin{aligned}\langle &inv(t_1, \mathtt{write}_1, (1,2)), \mathit{flush}(t_1, \bot), ret(t_1, \mathtt{write}_1, \bot), inv(t_1, \mathtt{read}_1, \bot),\\ &ret(t_1, \mathtt{read}_1, (1,2)), inv(t_2, \mathtt{read}_2, \bot), ret(t_2, \mathtt{read}_2, (0,0)),\\ &\mathit{flush}(t_1, \bot), \mathit{flush}(t_1, \bot), \mathit{flush}(t_1, \bot), \mathit{flush}(t_1, \bot), \mathit{flush}(t_1, \mathtt{write}_1)\rangle\end{aligned}$$

where t_1's local buffer is not fully flushed until after the two reads (there are 6 stores including the acquisition and release of the lock). $trans(h_{TSO})$ is then

$$\begin{aligned}\langle &inv(t_1, \mathtt{write}_1, (1,2)), inv(t_1, \mathtt{read}_1, \bot), ret(t_1, \mathtt{read}_1, (1,2)),\\ &inv(t_2, \mathtt{read}_2, \bot), ret(t_2, \mathtt{read}_2, (0,0)), ret(t_1, \mathtt{write}_1, \bot)\rangle\end{aligned}$$ □

The transformed history intuitively captures the behaviour on TSO and can be compared to histories of the abstract specification using the definition of linearizability of Section 2. Thus, linearizability on TSO is defined [8] by first transforming a concurrent history according to *trans*, then (as in the standard definition) comparing the result to an abstract history.

Definition 2 (TSO-linearizability). *An implementation of a concurrent object is linearizable on TSO with respect to a specification of the object if for each history h_{TSO} of the implementation, there exists a (sequential) history hs of the specification such that conditions of Definition 1 hold with $h = trans(h_{TSO})$.* □

4 Observational definition of linearizability on weak memory models

The approach to defining linearizability on TSO in Section 3.1 can be similarly applied to other weak memory models [7]. However, it depends on an understanding of the implementation details of the architecture which are used to derive the implementation histories. It also leads to a different relationship between concrete and abstract histories, specifically one involving a composition of a history transformation function and the standard definition of linearizability. This means existing proof techniques, and their support tools, need to be extended.

Adopting an observational, rather than architectural, definition of linearizability overcomes both of these problems. It abstracts from architectural details,

being based instead on memory order, and requires the standard linearizability relationship to hold between concrete and abstract histories.

In this section, we use the memory order $<_{m(p)}$ to define our framework for linearizability. To do so we begin by formalising the notion of an execution, which allows us to define the memory order in terms of the program commands.

4.1 Executions

On any memory model, an *execution* is a sequence of commands, which interact with shared memory. Branching statements (such as `if (condition)` and `while (condition)`) are included in executions as loads of the shared variables in `condition`, and their presence in a program affects the executions of that program. For example, letting $store(x, v)$ be a command which writes value v to variable x, and $load(x)$ be a command which reads x, the executions of the program fragment

```
x = n;
if (x > 0) y = n;
```

include $\langle store(x, 1), load(x), store(y, 1)\rangle$ when $n = 1$ and $\langle store(x, 0), load(x)\rangle$ when $n = 0$, but not $\langle store(x, 0), load(x), store(y, 0)\rangle$ when $n = 0$.

We now formalise what we mean by commands. A *command* is either an *invocation* of an operation, or a *load*, a *store*, an atomic *read-modify-write*, or a memory-model specific command. For example, for TSO we add *fence* and *flush* commands. We could add *returns* of operations to commands but instead identify the return of an operation with the last command associated with that operation. Each command is identified as being executed by a thread from type T, belonging to an occurrence of an operation of type I[4]. Invocations have an associated input from domain In. Load commands have an associated variable of type Var (the variable that they read) and write and read-modify-write commands have both an associated variable and a value of type Val (the value written to that variable). For example, the statement `x1 = d1` in operation `write` of seqlock when performed by a thread $t \in T$ is represented by the command $store(t, \texttt{write}_n, x1, d1)$.

$$\begin{aligned} Command \mathrel{\widehat{=}} & \; inv\langle\!\langle T \times I \times In\rangle\!\rangle \mid store\langle\!\langle T \times I \times Var \times Val\rangle\!\rangle \mid \\ & load\langle\!\langle T \times I \times Var\rangle\!\rangle \mid RMW\langle\!\langle T \times I \times Var \times Val\rangle\!\rangle \mid \ldots \end{aligned}$$

For a command c, we let $c.t \in T$ denote the thread that executed the command and $c.i \in I$ denote the operation it belongs to, and (where applicable) $c.var \in Var$ denote the variable of the command. We let the predicate $inv?(c)$ hold iff c is an invocation command, and $store?(c)$, $load?(c)$ and $rmw?(c)$ iff it

[4] To distinguish identical commands such as the two stores to `c` in the `write` operation of seqlock we would add other identifying information such as program counters, but we elide that detail here.

is a store, load or read-modify-write command, respectively. For TSO, we let the predicates *fence*?(c) and *flush*?(c) hold iff c is a fence or flush command.

We now define the executions *exec* for a program p on a sequentially consistent architecture, and those $exec_m$ on a memory model m. These are subsets of $exec_0$ which are the executions of p that can occur on any memory model that supports bypassing. First, an *Execution* is defined as a sequence of commands.

$$Execution \mathrel{\widehat{=}} \text{seq } Command$$

We then let *Object* denote the set of all concurrent objects. Such objects are represented by the implementation model (a state transition system) in the proof method of Derrick et al. [5, 6, 15]. For any $o \in Object$ and program p comprising a sequence of (potentially overlapping) calls to o's operations, let $exec_0(o, p)$ denote the set of executions of p obtained by *any* reordering of the commands of p that satisfy the following properties:

(a) If a certain command occurs within a branch of the program due to, for example, an `if` or `while` statement, the command should occur in an execution precisely when that particular branch is taken in the execution. This ensures the control structure of the program is respected in the reordered executions.
(b) Whenever a load r is moved before a store w to the same variable, the resulting executions behave as if the value read by r is that written by w. This captures the notion of bypassing introduced in Section 3. In a state transition-system approach (like that of Derrick et al. [5, 6, 15]) it could be captured by an additional variable for each thread t and shared variable x capturing the latest value written to x by t.

So $exec_0(o, p)$ contains all reorderings of p's commands that satisfy (a) and (b), and thus corresponds to those that can occur on any weak memory model that supports bypassing. Since bypassing is a common feature of weak memory models, we use this set of executions as the basis of our definitions. However, (b) could be dropped for a particular memory model if necessary.

The program order $<_p \subseteq Command \times Command$ of program p captures the order that the commands occur in the code run by each thread. An invocation command $inv(t, i, in)$, although not explicitly appearing in the code, is ordered as if it appeared in the code before the first statement of i, i.e., $inv(t, i, in) <_p c$ for all commands c of operations called by p with $c.t = t$ and $c.i = i$.

Note that $<_p$ is not a total order. It does not relate commands of different threads. We assume all synchronisation between threads (e.g., acquiring and releasing locks) is done in terms of loads and stores to shared variables. We don't actually formalise $<_p$ here although one could since a program can be formalised as a sequence of invocation commands, and then $<_p$ can be formalised in terms of the order of the invocations, program counters, etc., which define the program order for each thread.

The executions of a program p on a sequentially consistent architecture are precisely those executions which respect the program order $<_p$:

$$exec(o,p) \mathrel{\widehat{=}} \{e : exec_0(o,p) \mid \forall\, i,j : \mathrm{dom}\, e \bullet e(i) <_p e(j) \Rightarrow i < j\}$$

For a given memory model m, $<_{m(p)} \subseteq Command \times Command$ is a partial order on commands capturing the *memory order*. This order is generally weaker than the program order allowing reordering of certain commands (as in the table for TSO in Section 3). The executions of a program p on memory model m are those executions which respect the order $<_{m(p)}$:

$$exec_m(o,p) \mathrel{\widehat{=}} \{e : exec_0(o,p) \mid \forall\, i,j : \mathrm{dom}\, e \bullet e(i) <_{m(p)} e(j) \Rightarrow i < j\}$$

For all $c_1, c_2 \in Command$, the memory order for TSO is defined to maintain the program order unless the first command is a store and the second a load (the condition represented in the table in Section 3).

$$\begin{aligned} c_1 <_{TSO(p)} c_2 \;\Leftrightarrow\; & c_1 <_p c_2 \wedge (\neg\, (store?(c_1) \wedge load?(c_2)) \vee \\ & (\exists f : Command \bullet c_1 <_p f \wedge f <_p c_2 \wedge fence?(f))) \end{aligned}$$

The final predicate ensures we maintain program order if c_1 and c_2 are separated by a fence. Note that bypassing when the commands are to the same variable is covered by condition (b) above.

Example 3. Consider behaviour $\langle write(t_1,1,2);\ read(t_1,1,2);\ read(t_2,0,0)\rangle$ of seqlock consistent with a program p. For all commands c_1 and c_2 where $c_1.i = \texttt{write}_1$ and $c_2.i = \texttt{read}_1$, we have $c_1 <_{TSO(p)} c_2$. However, it is not the case that $c_1 <_{TSO(p)} c_2$ for any c_1 and c_2 where $c_1.t \neq c_2.t$. Considering just the operation $write(t_1,1,2)$, the second `c++` statement corresponds to commands $load(t_1, \texttt{write}_1, c)$ followed by $store(t_1, \texttt{write}_1, c, 2)$. These commands are preceded by the command $store(t_1, \texttt{write}_1, x2, 2)$ corresponding to the statement $x2 = d2$ (as $d2 = 2$). Hence, we have $store(t_1, \texttt{write}_1, x2, 2) <_p load(t_1, \texttt{write}_1, c) <_p store(t_1, \texttt{write}_1, c, 2)$. However, on TSO while $store(t_1, \texttt{write}_1, x2, 2) <_{TSO(p)} store(t_1, \texttt{write}_1, c, 2)$ and $load(t_1, \texttt{write}_1, c) <_{TSO(p)} store(t_1, \texttt{write}_1, c, 2)$, it is not the case that $store(t_1, \texttt{write}_1, x2, 2) <_{TSO(p)} load(t_1, \texttt{write}_1, c)$. □

4.2 Relating executions to histories

Histories can be derived from a set of executions as follows. Let out_o be a partial function which returns the output value produced on completion of the execution e on object o[5]. The domain of out_o will be those executions which end with the final command of an operation of o. The history corresponding to an execution e is then defined by $hist(e)$, where $last\ s$ is the last element of a sequence s, and $front\ s$ is the sequence s without the last element:

$$hist(e) \mathrel{\widehat{=}} \begin{cases} \langle\,\rangle & \text{if } e = \langle\,\rangle \\ hist(front\ e) \frown \langle last\ e\rangle & \text{if } inv?(last\ e) \\ hist(front\ e) \frown \langle ret((last\ e).t, (last\ e).i, out_o(e))\rangle & \text{if } e \in \mathrm{dom}\, out_o \\ hist(front\ e) & \text{otherwise} \end{cases}$$

[5] Since commands are deterministic there is exactly one such value.

4.3 Linearizability on a weak memory model

The observational definition of linearizability generalises that of Section 2. Whereas the concrete histories which the existing definition refers to are elements of $\{hist(e) \mid e \in exec(o,p)\}$, those for the observational definition are elements of $\{hist(e) \mid e \in exec_m(o,p)\}$, for a given memory model m.

Definition 3 (Linearizability on memory model m). *An implementation of a concurrent object o is linearizable on memory model m with respect to a specification of the object when, for any program p representing calls to the object, for each history in $\{hist(e) \mid e \in exec_m(o,p)\}$, there exists a (sequential) history hs of the specification such that the conditions of Definition 1 hold.* □

Note that the relationship between abstract and concrete histories in this definition are identical to that in Definition 1. Hence, there is no need to change the proof method or tool support. The memory model would be accounted for in the derivation of the implementation model of the approach of Derrick et al. [5, 6, 15], rather than in the simulation rules.

Below we show that, for TSO, Definition 3 is equivalent to Definition 2. Given $o \in Object$, we let $tso(o)$ denote the corresponding object on TSO, i.e., the object extended to include store buffers and flush commands for each thread (see [8] for one approach for doing this). The function *hist* that derives TSO histories from TSO executions is as in Section 4.2 with the addition of flush commands being retained.

Theorem 1. *Given $Hist(E) \mathrel{\widehat{=}} \{hist(e) \mid e \in E\}$ and $Trans(H) \mathrel{\widehat{=}} \{trans(h) \mid h \in H\}$, for any set of executions E and set of histories H:*

$$Trans(Hist(exec(tso(o), p))) \;=\; Hist(exec_m(o, p))$$

That is, the set of concrete histories related to abstract histories by the standard definition of linearizabilty are the same in each approach.

Proof For each $e \in exec(o,p)$ there will be a set of executions in $exec(tso(o),p)$. Each such execution e_{tso} is derived from e by adding flush commands such that the following holds.

$$\begin{array}{l}
\exists\, map_{tso} : \{i : \mathrm{dom}\, e_{tso} \mid store?(e_{tso}(i))\} \rightarrowtail\!\!\!\!\!\twoheadrightarrow \{i : \mathrm{dom}\, e_{tso} \mid flush?(e_{tso}(i))\} \bullet \\
\quad (\forall\, i : \mathrm{dom}\, map_{tso} \bullet \\
\qquad i < map_{tso}(i) \wedge \\
\qquad (\nexists j : \mathrm{dom}\, e_{tso} \bullet i < j < map_{tso}(i) \wedge (fence?(e_{tso}(j)) \vee rmw?(e_{tso}(j))))) \wedge \\
\quad (\forall\, i, j : \mathrm{dom}\, map_{tso} \bullet i < j \wedge e_{tso}(i).t = e_{tso}(j).t \Rightarrow map_{tso}(i) < map_{tso}(j))
\end{array}$$

That is, there exists a bijection mapping the positions of stores in e_{tso} and flushes in e_{tso} such that the flushes occur after the matching stores but before the next fence or read-write-modify command (which includes a fence as discussed in Section 3), if any, and the matching flushes for stores of a given thread t are in the order of the stores. The latter is due to the store buffer for a given thread being a FIFO queue.

Furthermore, all threads other than a given thread t run as if each store of t occurs at the point where the associated flush occurs.

Similarly, for each $e \in exec(o, p)$ there will be a set of executions in $exec_m(o, p)$. Each such execution e_m is derived by reordering commands, i.e, $items\ e_m = items\ e$ (where $items\ s$ is the bag of elements in the range of sequence s), such that the following holds.

$$\begin{array}{l}\exists\, map_m : \{i : \mathrm{dom}\, e \mid store?(e(i+1))\} \rightarrowtail\!\!\!\!\!\!\twoheadrightarrow \{i : \mathrm{dom}\, e_m \mid store?(e_m(i))\} \bullet \\ \quad (\forall\, i : \mathrm{dom}\, map_m \bullet \\ \qquad i < map_m(i) \wedge \\ \qquad (\nexists j : \mathrm{dom}\, e_m \bullet i < j < map_m(i) \wedge (fence?(e_m(j)) \vee rwm?(e_m(j))))) \wedge \\ \quad (\forall\, i, j : \mathrm{dom}\, map_m \bullet i < j \wedge e_m(i).t = e_m(j).t \Rightarrow map_m(i) < map_m(j))\end{array}$$

That is, there exists a bijection mapping the positions of the commands immediately preceding stores before reordering (i.e., in e) to the positions of the stores in the reordered execution e_m. The stores cannot be reordered with fences or read-modify-write commands, nor with other stores of the same thread. The latter ensures the order of the moved stores is the same as their original ordering.

In this case, all threads other than a given thread t run as if each store of t occurs at the point where the store is moved to (t runs as if the store occurs in its original position due to bypassing).

It can readily be deduced from the above that for any $e_{tso} \in exec(tso(o), p)$ there is an $e_m \in exec_m(o, p)$, and vice versa, such that the executions include the same commands (apart from flush commands) and for a given store command s, the position of s's flush with respect to other commands in e_{tso} is the position of s with respect to other commands in e_m.

Applying *hist* to e_{tso} and then *trans* to the resulting history gives us a history h_{tso} where the return of each operation is either the last command of that operation, or a flush associated with a store of the operation, whichever occurs later in e_{tso}.

Applying *hist* to e_m gives us a history h_m where the return of each operation is the last command of that operation which may be a store which has been moved forward in the execution beyond the original last command of the operation.

Hence, due to the correspondence between positions of flushes in e_{tso} and the positions that stores are moved to in e_m it follows that $h_{tso} = h_m$. Therefore, $Trans(Hist(exec(tso(o), p))) = Hist(exec_m(o, p))$ as required. □

Example 4. Consider the concurrent object in Figure 3 and a program p on the object in which a thread t_1 calls `OpOne` and a thread t_2 calls `OpTwo`. If the values stored to `x` and `y` are not flushed until the end, then one can observe the following behaviour on TSO: $\langle OpOne(t_1, 1, 0);\ OpTwo(t_2, 0, 1)\rangle$. The table below captures the program order of p, and the memory order that results. (Commands are abbreviated to their essential components for readability.)

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

OpOne(out word x1, y1) {
   x=1;
   x1=x;
   y1=y;
}
```

```
OpTwo(out word x1, y1) {
   y=1;
   y1=y;
   x1=x;
}
```

Fig. 3. Simple concurrent object

Name	Program order of p	Memory order for TSO
inv_1	$inv(t_1)$	
s_1	$store(t_1, x, 1)$	$inv_1 <_{TSO(p)} s_1$
l_1	$load(t_1, x)$	$inv_1 <_{TSO(p)} s_1$, $inv_1 <_{TSO(p)} l_1$
l_2	$load(t_1, y)$	$inv_1 <_{TSO(p)} s_1$, $inv_1 <_{TSO(p)} l_1 <_{TSO(p)} l_2$
inv_2	$inv(t_2)$	
s_2	$store(t_2, y, 1)$	$inv_2 <_{TSO(p)} s_2$
l_3	$load(t_2, y)$	$inv_2 <_{TSO(p)} s_2$, $inv_2 <_{TSO(p)} l_3$
l_4	$load(t_2, x)$	$inv_2 <_{TSO(p)} s_2$, $inv_2 <_{TSO(p)} l_3 <_{TSO(p)} l_4$

From this one can construct valid executions that respect program order, e.g., $e = \langle inv_1, s_1, l_1, l_2, inv_2, s_2, l_3, l_4 \rangle$. An associated execution on TSO which corresponds to the behaviour $\langle OpOne(t_1, 1, 0); OpTwo(t_2, 0, 1) \rangle$ is $e_{tso} = \langle inv_1, s_1, l_1, l_2, inv_2, s_2, l_3, l_4, flush(t_1, \texttt{OpOne}_1), flush(t_2, \texttt{OpTwo}_1) \rangle$. The corresponding execution respecting memory order is $e_m = \langle inv_1, l_1, l_2, inv_2, l_3, l_4, s_1, s_2 \rangle$. The histories of e_{tso} and e_m can then be calculated as follows:

$$\begin{aligned} hist(e_{tso}) = \langle & inv(t_1, \texttt{OpOne}_1, \bot), ret(t_1, \texttt{OpOne}_1, (1, 0)), inv(t_2, \texttt{OpTwo}_1, \bot), \\ & ret(t_2, \texttt{OpTwo}_1, (0, 1)), flush(t_1, \texttt{OpOne}_1), flush(t_2, \texttt{OpTwo}_1) \rangle \\ hist(e_m) = \langle & inv(t_1, \texttt{OpOne}_1, \bot), inv(t_2, \texttt{OpTwo}_1, \bot), ret(t_1, \texttt{OpOne}_1, (1, 0)), \\ & ret(t_2, \texttt{OpTwo}_1, (0, 1)) \rangle \end{aligned}$$

Then it is easy to see that $trans(hist(e_{tso}))$ is the same as $hist(e_m)$. □

5 Generalising to other memory models

The novelty of the work presented here is that it allows the definition of linearizability to be easily applied to other other well understood architectures. One example that is easy to illustrate is the Partial Store Order (PSO). PSO essentially mimics TSO except with one additional relaxation, namely that PSO only guarantees stores to the same variable are in order whereas stores to different variables may be reordered. Hence, for all $c_1, c_2 : Command$, its memory order on program p is

$$\begin{aligned} c_1 <_{PSO(p)} c_2 \;\Leftrightarrow\; & c_1 <_p c_2 \wedge \\ & (\neg\, (store?(c_1) \wedge load?(c_2) \vee \\ & \quad store?(c1) \wedge store?(c2) \wedge c1.var \neq c2.var) \vee \\ & \quad (\exists f : Command \bullet c_1 <_p f \wedge f <_p c_2 \wedge fence?(f))) \end{aligned}$$

The Power and ARM architectures are more complex. As well as allowing reordering of commands these architectures (1) are *non-multiple-copy-atomic*, meaning a write by one thread may propagate to the other threads at different times, and (2) allow *speculative execution*, where statements after a branch command may be executed before the branch condition has been determined (and executions of paths not subsequently followed discarded).

The former can be incorporated into our framework by allowing each thread to have its own copy of the global variables in the implementation model (as suggested in [13]). The latter can be incorporated by enabling the implementation model to nondeterministically decide on a branch condition at any time and then terminate when the decision is found to be incorrect. Since speculative execution does not effect the external behaviour of a concurrent object before the branch point is reached, no new behaviour is introduced when an execution is terminated at the branch point. The nondeterminism ensures all possible speculative executions are considered, including those which do not terminate at the branch point.

Like command reordering, the above behaviours of Power and ARM can be observed via systematic testing [13]; they do not require an architectural understanding of the memory model. Incorporating them into our framework is an ongoing area of work.

In addition to providing a general definition of correctness for a variety of memory models, the framework allows us to use the existing simulation rules and tool support for linearizability. Specifically, since the framework does not include architectural details in the model derived from the code we do not need new simulation rules or tool support for each memory model. All that is needed is the ability to derive implementation models from code. This is an important area of future work. In Derrick et al. [5], sequentially consistent executions are derived from a state transition system in which each transition corresponds to a command and is enabled precisely when a program counter variable pc, for a given thread, is set to the line number of that command in the program code. To allow reordering, we would need to allow certain commands to be able to occur over a range of values of pc, while respecting additional constraint on the relative order of their occurrence with other commands such as fences.

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. 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, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.*, 33(1):4:1–4:43, 2011.
6. 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.
7. J. Derrick and G. Smith. A framework for correctness criteria on weak memory models. In N. Bjørner and F.S. de Boer, editors, *FM 2015*, volume 9109 of *LNCS*, pages 178–194. Springer, 2015.
8. J. Derrick, G. Smith, and B. Dongol. Verifying linearizability on TSO architectures. In E. Albert and E. Sekerinski, editors, *iFM 2014*, volume 8739 of *LNCS*, pages 341–356. Springer, 2014.
9. 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.
10. J. Fitzpatrick. An interview with Steve Furber. *Commun. ACM*, 54(5):34–39, 2011.
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. Maranget, S. Sarkar, and P. Sewell. A tutorial introduction to the ARM and POWER relaxed memory models. Draft available from http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf, 2012.
14. W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In *Automated Deduction*, pages 13–39. Kluwer, 1998.
15. G. Schellhorn, H. Wehrheim, and J. Derrick. A sound and complete proof technique for linearizability of concurrent data structures. *ACM Trans. on Computational Logic*, 15(4):31:1–31:37, 2014.
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. G. Smith, J. Derrick, and B. Dongol. Admit your weakness: Verifying correctness on TSO architectures. In I. Lanese and E. Madelaine, editors, *FACS 2014*, volume 8997 of *LNCS*, pages 364–383. Springer, 2014.
18. 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.
19. O. Travkin, A. Mütze, and H. Wehrheim. SPIN as a linearizability checker under weak memory models. In V. Bertacco and A. Legay, editors, *HVC2013*, volume 8244 of *LNCS*, pages 311–326. Springer, 2013.
20. O. Travkin and H. Wehrheim. Handling TSO in mechanized linearizability proofs. In E. Yahav, editor, *HVC2014*, volume 8855 of *LNCS*, pages 132–147. Springer, 2014.
21. V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.