Modelling concurrent objects running on the TSO and ARMv8 memory models

Kirsten Winter*, Graeme Smith

School of Information Technology and Electrical Engineering, The University of Queensland

John Derrick

Department of Computing, University of Sheffield

Abstract

Hardware weak memory models, such as TSO and ARM, are used to increase the performance of concurrent programs by allowing program instructions to be executed on the hardware in a different order to that specified by the software. This places a challenge on the verification of concurrent objects used in these programs since the variations in the executions need to be considered.

Many approaches exist for verifying concurrent objects along with associated tool support. In particular, we focus on a thread-local approach to checking *linearizability*, the standard correctness condition for concurrent objects, using a model checker. This approach, like most others, does not support weak memory models. In order to reuse this existing approach, therefore, we show how to use the semantics of a weak memory model to directly derive a transition system of concurrent objects running under it.

We do this for both TSO and the latest version of ARM, ARMv8. Since there is a straightforward implementation of TSO, we reflect this in our transition system which includes a buffer of writes to memory mirroring the store buffer of TSO. We illustrate linearizability checking using model checking on a transition system generated by this approach.

The implementation of the significantly more complex ARMv8 architecture is less obvious. We derive our transition system in this case from an exisiting operational semantics that is consistent with the results of thousands of litmus test run on ARM hardware.

Keywords: linearizability, weak memory models, TSO, ARMv8, model checking

*Corresponding author
Email address: `kirsten@itee.uq.edu.au` (Kirsten Winter)

1. Introduction

Hardware weak memory models are used on all modern computing platforms. TSO (supported by Intel and AMD processors) [1, 2] is used in nearly all laptop and desktop computers, ARM [3, 4] is used in most mobile devices, and IBM POWER [5] on IBM servers and supercomputers. These memory models are aimed at increasing the performance of programs running on them by limiting software control of when accesses to the global memory occur. Instead such accesses are placed under the hardware's control, and may occur out-of-order with respect to the order they occur in the program text.

Weak memory models such as those of the processors mentioned above, guarantee that any instruction ordering they allow does not change a program's behaviour, provided the program is data-race free. However, many concurrent objects, i.e., objects designed to be utilised by multiple threads [6], utilise non-blocking algorithms which are inherently racy. This complicates the verification of concurrent objects.

The standard notion of correctness for concurrent objects is *linearizability* [7]. Over the years, a number of approaches have been developed for proving linearizability along with associated tool support [8, 9, 10, 11, 12, 13, 14, 15]. In particular, Derrick et al. [13] provide a *thread-local*, *step-local* proof method supported by the theorem prover KIV [16]. Being thread-local, correctness for objects being accessed by an arbitrary (even infinite) number of threads follows from results on the object being accessed by a single thread. Being step-local means these results can be obtained by proofs on one step, i.e., one program instruction, at a time. These features make the approach amenable to model checking without restrictions on the number of threads or number of steps that other model checking approaches need to make [17].

Like the other approaches cited above, Derrick et al.'s approach assumes that the concurrent objects are running on a *sequentially consistent* architecture, i.e., one where instructions, e.g., loads to and stores from global memory, take effect in the order they appear in the program. Hence, it cannot be directly applied to objects on weak memory models.

In this article, we provide a means of using both the proof method of Derrick et al. and its model checking support on hardware weak memory models. Specifically, we develop means of generating transition systems from the code of concurrent objects capturing weak memory effects for both TSO and the latest version of ARM, ARMv8 [3].[1]

We begin in Section 2 by introducing the concept of linearizability and our running example, the Linux reader-writer mechanism seqlock [18]. We also describe in more detail the proof method of Derrick et al. In Section 3 we discuss weak memory models using TSO as an example, and in Section 4 we describe the general approach for generating a transition system. We provide a specific

[1]By ARMv8 we refer to the multi-copy atomic revision [3], not the original ARMv8 processor which was non-multi-copy atomic [4].

approach for TSO in Section 5 and then in Section 6 apply it to seqlock and summarise our model checking results using NuSMV [19]. In Section 7 we provide the approach for generating a transition system for a concurrent object on ARMv8. Unlike TSO, the full details of an implementation of ARM's architecture cannot easily be derived from the existing documentation. Hence, the model is derived from an existing operational semantics [20, 21] that is consistent with the results of thousands of litmus test run on ARM hardware. In Section 8 we revisit the seqlock example under ARMv8 before concluding the paper in Section 9.

Contributions. This paper is an extension of our earlier work [22]. That paper presented models for TSO and XC (a theoretical weak memory model allowing instruction reorderings similar to ARM) [2]. The latter was presented as a stepping stone to more complex memory models like ARM. To accurately model ARMv8 in this paper, we take into account various complex features not supported in XC including speculative execution, control fences, address shifting, load speculation and write elimination.

2. Linearizability

Concurrent objects are objects that are developed to be used in a multi-threaded environment [6]. Generally, they allow more than one thread to access them simultaneously. Consider, for example, the Linux reader-writer mechanism *seqlock*, which allows reading of shared variables without locking the global memory, thus supporting fast write access. A thread wishing to *write* to the shared variables `x1` and `x2` acquires a software lock (by atomically setting a variable `lock` to 0 when it is 1)[2] and increments a counter `c`. It then proceeds to write to the variables, and finally increments `c` again before releasing the lock (by setting `lock` to 1). The lock ensures synchronisation between writers, and the counter `c` ensures the consistency of values read by other threads. 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.

An abstract specification of seqlock, in which the read and write operations are regarded as atomic, is given in Figure 1. A typical implementation, in which the statements of operations may be interleaved, is given in Figure 2. In the implementation, 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`.

Linearizability [7] is the standard correctness criterion for verifying concurrent objects such as seqlock. It is used to relate each *history*, i.e., allowed

[2]This can be implemented, for example, using a spin lock [23].

```
x1 = 0, x2 = 0;

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

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

Figure 1: seqlock specification

```
x1 = 0, x2 = 0;
c = 0, lock = 1;

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

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

Figure 2: seqlock implementation (from [24])

sequence of operation invocations and returns, of a concurrent implementation to a matching *sequential history* of a specification in which all operations are regarded as atomic. It does this based on the understanding that each operation in the implementation can be viewed as taking effect instantaneously at some point between its invocation and return; a point known as the *linearization point.* For example, in the implementation of 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. The key consequence of the definition of linearizability is that if two concrete operations overlap (due to concurrency), then they may take effect in any order from an abstract perspective, but otherwise they must take effect in the order in which they are invoked.

A formal definition of linearizability is given in [7] and a number of approaches have been developed for proving it along with associated tool support [8, 9, 10, 11, 12, 13, 14, 15]. In particular, Derrick et al. [13] have developed a simulation-based proof method for linearizability which is both *thread-local*, i.e., reasoning is performed on a single thread, and *step-local*, i.e., reasoning is performed on one line of code at a time.

The proof method of Derrick et al. is based on the idea that if an implementation of an operation C is linearizable it simulates the behaviour of the abstract specification of that operation, A. All intermediate states of C (between its lines of code) must be related to either the pre- or post-state of A

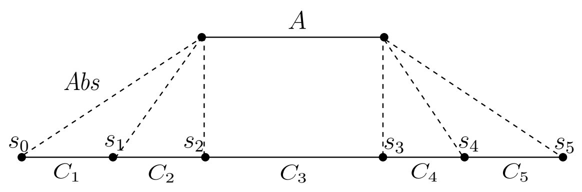

Figure 3: Simulation-based proof method for linearizability

via an abstraction relation Abs. In Figure 3, for example, the step C_3 matches the state change of A whereas all other steps of C match abstract skips on the pre-state or the post-state of A. To enable step-local proofs, the states of C are labelled with assertions s_i (stating the required conditions at that point of the execution) that the program needs to maintain, i.e., when executed in a state where s_i holds, step C_{i+1} must lead to a state where s_{i+1} holds.

In the following discussion of the method, we let AS be the state space of the specification, and GS and LS be the state spaces comprising the global and local variables of the implementation, respectively. Also, we let $gs, gs' \in GS$ denote the global variables before and after an operation, respectively, and similarly $ls, ls' \in LS$ the local variables before and after an operation, and $as, as' \in AS$ the abstract state before and after an operation.

The aforementioned assertions on the implementation state are collected into an invariant $Inv(gs, ls)$, i.e., for each line of code $Inv(gs, ls)$ includes a conjunct of the form $pc = i \Rightarrow s_i$, where $pc \in LS$ is the program counter, and s_i is the assertion that must hold at $pc = i$ in order for the implemented operation to simulate the abstract operation.

A function $status(gs, ls)$ is defined to identify the linearization point. The return type of this function is $STATUS == IDLE \mid IN\langle\langle In\rangle\rangle \mid OUT\langle\langle Out\rangle\rangle$, where statuses IN and OUT are parameterised by an element of the sets In, denoting all input values, and Out, denoting all output values, respectively. Before invocation, $status(gs, ls)$ is $IDLE$. After invocation but before the linearization point it is equal to $IN(in)$, where $in \in In$ is the input to the abstract operation, and after the linearization point it is equal to $OUT(out)$, where $out \in Out$ is the output of the abstract operation. The types In and Out have a special value $\perp$ denoting no input or output, respectively. As well as identifying the linearization point, the $status$ function is used to store the input of the invocation step until it is needed at the linearization point, and to store the abstract output of the linearization point until it is need at the return step.

Let σ and σ' be status values, and λ be a list of parameters comprising gs, gs', ls and ls', and possibly in or out. For a step C which is not the linearization point, the proof obligation is of the following form.[3]

[3]The proof obligation is slightly different when the step is the final step of an operation, needing to ensure the value returned by the step matches that stored in $status(gs, ls)$.

$$\begin{array}{l}\forall\, as : AS;\; gs, gs' : GS;\; ls, ls' : LS;\; in : In;\; out : Out \cdot \\ \quad Abs(as, gs) \land Inv(gs, ls) \land status(gs, ls) = \sigma \land C(\lambda) \Rightarrow \\ \qquad status(gs', ls') = \sigma' \land Abs(as, gs') \land Inv(gs', ls') \end{array} \tag{1}$$

That is, the step preserves the abstraction relation and invariant, but may change the status, e.g., if the step is an invocation step it changes the status from *IDLE* to $IN(in)$.

The step corresponding to the linearization point must simulate the abstract operation A. The proof obligation is of the following form.[4]

$$\begin{array}{l}\forall\, as : AS;\; gs, gs' : GS;\; ls, ls' : LS;\; in : In \cdot \\ \quad Abs(as, gs) \land Inv(gs, ls) \land status(gs, ls) = \sigma \land C(\lambda) \Rightarrow \\ \qquad (\exists\, as' : AS;\; out : Out \cdot A(in, as, as', out) \land \\ \qquad\quad status(gs', ls') = \sigma' \land Abs(as', gs') \land Inv(gs', ls')) \end{array} \tag{2}$$

That is, as well as preserving the abstraction relation and invariant, and changing status (from either *IDLE* or $IN(in)$ to either *IDLE* or $OUT(out)$), the step ensures that the implementation state change and inputs match those of A.

To ensure threads do not interfere with each other's behaviour, an additional proof step checks that each step does not change global variables in such a way that any of the required assertions for another thread can be broken.[5] This step is identical to the check of non-interference in the proof method of Owicki-Gries [25].

This amounts to showing that a thread p (with local state ls) cannot invalidate the invariant $Inv(gs, lsq)$ or change the status $status(gs, lsq)$ which another thread q (with local state lsq) relies on. To do this we require a further invariant $D(ls, lsq)$ relating the local states of two threads. For seqlock, this invariant includes a predicate that only one thread can be at lines 2 to 6 (due to the need to acquire the lock). That is, D includes the conjunct $pcq \in 2..6 \Rightarrow \neg\, pc \in 2..6$.

The proof obligation then requires we prove

$$\begin{array}{l}\forall\, as : AS;\; gs, gs' : GS;\; ls, ls', lsq : LS;\; in : In;\; out : Out \cdot \\ \quad Abs(as, gs) \land Inv(gs, ls) \land Inv(gs, lsq) \land D(ls, lsq) \land C(\lambda) \\ \qquad \Rightarrow Inv(gs', lsq) \land D(ls', lsq) \land status(gs', lsq) = status(gs, lsq) \end{array} \tag{3}$$

That is, the step preserves the invariant and status of the other thread q, as well as the invariant D.

Additionally, we have a proof obligation related to initialisation. Let *GSInit* and *LSInit* be the initial configurations of global variables and local variables respectively, and *ASInit* the initial configuration of the abstract state. The obligation ensures that the abstraction relation, invariant and D hold initially.

[4]The proof obligation is slightly different when the step is the final step of an operation, needing to ensure that the value returned by the step is also one allowed by $A(in, as, as', out)$.

[5]Note that all threads execute the same code (that of the concurrent object) and have the same program steps and assertions.

$$\begin{aligned}&\forall\, gs : GSInit \cdot \exists\, as : ASInit \cdot Abs(as, gs) \wedge \\ &\qquad (\forall\, ls : LSInit \cdot Inv(gs, ls)) \wedge (\forall\, ls, lsq : LSInit \cdot D(ls, lsq))\end{aligned} \tag{4}$$

Other than initialisation, each of these proof obligations is step-local, involving a single line of code, and changes the state of one thread. Together they have been shown to prove linearizability between the abstract and concrete specifications [13]. Hence the approach, carried out on a single thread, proves linearizability for an arbitrary number of threads accessing the concurrent object.

The proof method is supported by the KIV theorem prover [13] and, being thread-local and step-local, lends itself to automation using a model checker [17]. Unlike other model checking approaches for linearizability, the results are not restricted to a fixed number of threads, or particular sequences of operation calls. However, the approach is not fully automatic. As with the approach of Derrick et al. [13], the user must provide the invariants *Inv* and *D*. While the derivation of the former can be partially automated (only the assertion that holds in the idle state between operation calls needs to be provided) [26], the latter is derived from the user's understanding of how the code works.

3. Weak memory models

Existing proof methods for linearizability, such as Derrick et al.'s, are not directly applicable to objects running on a weak memory model. We explain this via the example of the well understood TSO architecture [1, 2].

In TSO, each core (hosting one or more threads) uses a *store buffer*, which is a FIFO queue that holds pending *stores* (i.e., writes) to memory. When a thread running on a 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*, committing (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 its core's local buffer, and only comes from the memory if the buffer is empty. This is referred to as *bypassing* in TSO. The use of local buffers can cause unexpected behaviour, e.g., a load by one thread, occurring after a store by another, may return an older value, behaving as if it occurred before the store.

In general, flushes are controlled by the hardware, and from the programmer's perspective occur nondeterministically. However, a programmer may explicitly include a *fence* instruction to force flushes to occur.

A typical situation is illustrated for seqlock in Figure 4 where the horizontal lines represent the execution of an operation (from its invocation to its response) and the vertical lines represent linearization points. The figure shows Thread 1 doing a write with values 1 and 2 followed by Thread 2 doing a read before Thread 1's writes are flushed by the "hardware thread". The read will return the initial values of `x1` and `x2`, which we assume to be 0 in the figure.

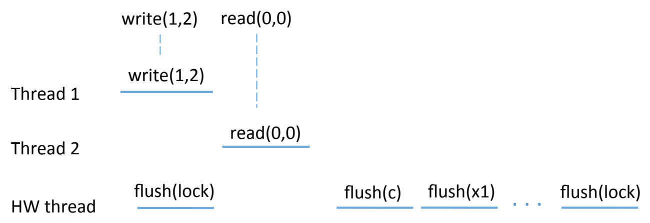

Figure 4: Linearizability fails on TSO

It has been argued that linearizability should succeed for this execution since the write operation remains active (and hence may take effect) up to the time of the flush of its final written value [24, 27, 28]. Hence, the write operation can linearize after the read. This is illustrated in Figure 5 where the occurrence of the write is extended to its final flush.

However, since the write and read do not overlap, the only matching specification history is one where the read occurs after the write. This is not allowed by the specification of seqlock and hence applying Derrick et al.'s proof method directly to the specification and code in Figures 1 and 2, linearizability cannot be proved. This is because the method assumes the code is running on a sequentially consistent architecture. To use such existing methods we propose building models of the concurrent objects which take the memory model into account, i.e., allow the additional behaviour that can occur under these memory models.

The behaviours of our models can be compared directly using any existing approach for linearizability. This is in contrast to earlier approaches such as [28]. In that work, one has to apply a transformation to each behaviour that encoded the effects of the TSO architecture. Only then can standard linearizability be used to check correctness [29]. We discuss how to construct our models for both TSO and ARMv8 in the following sections.

4. Modelling concurrent objects

The order of statements in a program defines the *program order*. On a sequentially consistent architecture, this defines the *execution (or memory) order*. However, in a weak memory model the execution order is different from the program order because statements can be reordered.

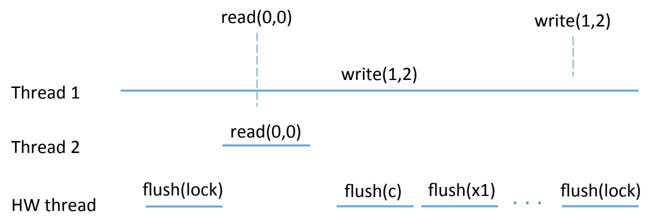

Figure 5: Linearizability succeeds on TSO

The method we describe generates a transition system that models the behaviour of a single thread on which any sequence of operations of a concurrent object may be called. The order in the transition system is the execution order, as opposed to the program order. The thread local nature of the proof obligations we need to verify means that this will be sufficient and enables an efficient approach to model checking.

Although such a thread invokes operations one after another, under a weak memory model instructions from successive operations may interleave. For example, on TSO a write that is buffered during one operation may be flushed during the following operation (and hence appear as if it occurred during the execution of that following operation).

We derive the order in the transition system directly from the semantics of the weak memory model under consideration. For example, for TSO we use the semantics given in [2] and for ARMv8 a semantics given in [20, 21]. While the semantics in [2] is based on an understanding of the TSO architecture (in particular, the use of store buffers), that in [20, 21] is consistent with the litmus tests run on ARM hardware.

In the programming model we consider, our statements are *loads*, *stores*, *branch* statements and *return* statements. We also include *invocations*, as well as architecture specific statements like *fences* and atomic *read-modify-writes (RMWs)* (e.g., the *compare-and-swap* (CAS) instruction [2]).

Branch statements are often not treated as instructions but simply as control flow directives [2, 1]. However, to model more complex weak memory models, like ARMv8 (where speculative execution is supported), it seems beneficial to treat branching as a statement (and hence as a separate transition) [20, 21].

Each program statement (labelled by a line number) is modelled as a state transition in a standard fashion. Statements that modify a global variable are split into two steps, one for loading the variable into a register (local to the thread) and modifying it, and one for storing the result back into the global variable (e.g., $c{+}{+}$ is modelled as $local_c = c + 1;\ c = local_c$, where $local_c$ is a register). Consequently, each assignment accesses at most one global variable. This ensures we allow all possible interference between threads.

Additionally, modifications on a register are merged with a corresponding statement affecting a global variable (e.g., $local_c = c;\ local_c = local_c + 1$ is modelled as $local_c = c + 1$). This simplifies our transition system in cases where interference does not matter.

We also model invocations as separate transitions. Although instructions from successive operations can be interleaved (on a single thread), the invocation of the second operation cannot happen until the first has returned. We model this explicitly in our transition systems.

Under a sequentially consistent architecture the transitions follow the program order prescribed by the code, i.e., a control flow graph $<_P = (\mathcal{L}_P, next_P)$ over the set of instruction labels $\mathcal{L}_P$ (i.e., line numbers) and a next-step relation $next_P : \mathcal{L}_P \times Bool \to \mathcal{L}_P$, which maps each label to one or more successors. Starting from 0, $next_P$ will deliver the next line number to be executed. In the case of a branch, the second argument determines which line number that is.

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

Table 1: Order constraints for TSO architectures [2]

When a line number has only one successor this second argument equals true and is omitted.

The root node of $<_P$ is 0 which models the idle state, and is the successor for the labels of return statements. In line 0, every operation can be invoked. We assume $next_P(0)$ returns the appropriate line number for the invoked operation. For clarity of presentation, we do not model this explicitly but it is included in our model checker implementation.[6]

Under weak memory models the (observed) order of transitions is weaker than $<_P$. This weaker order can be described as a set of control flow graphs where each is a restriction of $<_P$ to a subset of labels whose order must be maintained on the memory model. Instructions corresponding to line numbers in different graphs can be reordered with respect to each other.

In the following sections, we consider the TSO and ARMv8 memory models based on the semantics in [2] and [20, 21], respectively. For each we provide a set of rules for translating a given concurrent object into a transition system that is consistent with the memory model and sufficient for verifying the program's correctness.

5. A model for TSO

In [2], Sorin et al. describe the effects of TSO as a reordering of statements as summarised in Table 1. In the table, X denotes an enforced sequence of commands and B denotes that commands can be reordered but *bypassing* is required if the commands are to the same variable (see Section 3). From the information in this table, we can provide an explicit model of a concurrent object's behaviour under TSO.

From the table we can see that stores may be reordered with subsequent loads, but their results will be locally visible in program order. To capture this in our model, each store command is captured by two separate steps: a *local store* (*l-store*) which copies the value to be stored to a local register variable and a *global store* (*g-store*) which stores the register value to the corresponding

[6]See `http://staff.itee.uq.edu.au/kirsten/LinModels/SeqLock.html`.

global variable. *l-stores* follow the program order while *g-stores* follow a separate order but have to come after their corresponding *l-stores*. This enables us to model *bypassing* as well as the delayed observation of store steps.

An RMW statement is atomic and hence needs to write to memory immediately. Therefore, it necessarily includes a fence on TSO (since its write will be placed at the end of the FIFO store buffer) which prevents reordering. We assume that branch instructions (which are not included in the table) are not reordered with respect to loads, fences and RMWs, nor with each other, and that stores can be reordered after subsequent branches. This assumption is consistent with other models of TSO, e.g., [30, 31].

To reflect the given reordering constraints we introduce two (control flow) graphs ordering the labels of a program.

- a *load order* $<_L$ which orders steps with a local effect; it is identical to $<_P$, i.e., $<_L = <_P$
- a *store order* $<_S$ which orders steps with a global effect like *g-stores*, fences and RMWs; it also includes branches, as the branching structure in the load order need to be reflected in the store order, as well as invocations, since they cannot be reordered *after* stores: $<_S = (\mathcal{L}_S, next_S)$ where $\mathcal{L}_S$ includes all labels apart from those of loads.

5.1. Transition System Model

The order of transitions in a transition system can be enforced by a counter whose values relate unambiguously to each of the steps (e.g., the line number in the program). If the transition occurs the counter gets increased to the next value in the order, thereby enabling the next transition in the prescribed order.

To follow this standard way of encoding transition systems we introduce counters for the two orders: a load counter pc_L which ranges over all labels included in $<_L$, and a sequence of store counters, pc_S, in which each entry ranges over all labels in $<_S$. A sequence of store counters is required to represent overlapping invocations of the operation being modelled (see Section 5.2). Each transition is guarded by the counter(s) that enforces the order(s) which they are part of.

We let v_l be a local variable and v_g be a global variable. We also let r_g be a sequence of locally stored values to v_g in order of their occurrence, and for each branch we have a flag b_n indicating whether the branch condition at line n evaluated to true of false. This flag b_n is used to direct the flow in $<_S$ along the branch followed by the instructions in $<_L$. We also have a flag r_n for an *RMW* at line n. It is set to true when the *RMW* succeeds and false otherwise.

Initially, $pc_L = 0$ and $pc_S = \langle\,\rangle$. The transitions for each statement type take the following form.

5.2. Invocations and returns

When an operation is invoked it is possible that some *g-stores* of the previous operation have not occurred yet. Hence, pc_S is a sequence of store counters, each representing the store step of one operation. If a *g-store* from a previous occurrence of an operation corresponds to that operation's linearization point then, to apply the proof method of Derrick et al. described in Section 2, we need to have access to the previous operation's input and output values. Hence, the inputs and outputs of operations are also modelled as sequences of values (a value for each operation occurrence).

To invoke an operation we require that the invoking thread is idle, i.e., $pc_L = 0$. We extend the sequence of pc_S by another element and increase both the load and the (newly added) store counter to the first line number in the respective order graph for that operation. (Since there is only one successor node, the second argument of the $next_L$ function is *true* and is omitted here.)

We also extend the sequence of input values *in* with the operation's inputs. The inputs required for any step of the transition system are those last added to *in*, denoted $last(in)$. Note that *g-stores* (which may be from a previous invocation) do not refer to inputs, only local registers (as detailed in Section 5.3).

$$\begin{aligned} Invoke(\mathtt{op}(\mathtt{val_1}, \ldots, \mathtt{val_n})) == {} & pc_L = 0 \wedge \\ & pc'_L = next_L(0) \wedge \\ & pc'_S = pc_S \frown \langle next_S(0) \rangle \wedge \\ & in = in \frown \langle (val_1, \ldots, val_n) \rangle \end{aligned}$$

A return step, $\mathtt{n} : \mathtt{return}(\mathtt{val_1}, \ldots, \mathtt{val_n})$, of an operation adds the output values to the sequence *out*. When there is no explicit return statement (as in the write operation of seqlock), the final statement of each execution of the operation is treated as a return statement updating *out* with $\perp$, indicating no output.

$$\begin{aligned} Return(\mathtt{n} : \mathtt{return}(\mathtt{val_1}, \ldots, \mathtt{val_n})) == {} & pc_L = n \wedge \\ & pc'_L = 0 \wedge \\ & out' = out \frown \langle (val_1 \ldots val_n) \rangle \end{aligned}$$

5.3. Loads and stores

A load can occur when pc_L has reached its line number, and upon occurrence it sets pc_L to the next line number in the load order. The value to be loaded might be found either at the end of the corresponding register r_g (if an *l-store* for that variable has occurred but not the corresponding *g-store*) or in the global memory, namely v_g. Assume $last(s)$ denotes the last element of a sequence s. Then given a load $\mathtt{n} : \mathtt{v_l} = \mathtt{e}(\mathtt{v_g})$ (where $e(v_g)$ is an expression in terms of v_g modelling the merge of a load and possibly a modification to the loaded value), we have the following transition.

$$\begin{aligned} Load(\mathtt{n} : \mathtt{v_l} = \mathtt{e}(\mathtt{v_g})) == {} & pc_L = n \wedge \\ & pc'_L = next_L(n) \wedge \\ & v'_l = (\textbf{if } r_g = \langle\,\rangle \textbf{ then } e(v_g) \textbf{ else } e(last(r_g))) \end{aligned}$$

An *l-store* can occur when pc_L has reached the line number of a store. Upon occurrence it sets pc_L to the next line number in the load order and appends the new value to the corresponding register variable r_g. Given the line $\mathtt{n} : \mathtt{v_g} = \mathtt{val}$ we have

$$\begin{aligned} \mathit{Store}(\mathtt{n} : \mathtt{v_g} = \mathtt{val}) == \ & pc_L = n \land \\ & pc'_L = next_L(pc_L) \land \\ & r'_g = r_g \frown \langle val \rangle \end{aligned}$$

A *g-store* can occur when its label is at the head of the sequence pc_S and if the register r_g is not empty, i.e., the corresponding *l-store* has occurred. When a *g-store* occurs it removes the value at the head of the register r_g and writes it to global memory (i.e., updates v_g). It also updates the head of pc_S to $next_S(n)$ unless $next_S(n) = 0$ (i.e., the stores of the current operation are finished), in which case the step removes the head of pc_S (so that the first *g-store* of the next operation can occur if there is any).

$$\begin{aligned} \mathit{GStore}(\mathtt{n} : \mathtt{v_g} = \mathtt{val}) == \ & head(pc_S) = n \land r_g \neq \langle\rangle \land \\ & v'_g = head(r_g) \land \\ & r'_g = tail(r_g) \land \\ & pc'_S = (\textbf{if } next_S(n) \neq 0 \\ & \qquad \textbf{then } \langle next_S(n) \rangle \frown tail(pc_S) \\ & \qquad \textbf{else } tail(pc_S)) \end{aligned}$$

5.4. Branches

A branch or loop instruction[7] is captured by two transitions, one *BranchT* for when the branch condition is true, and the other *BranchF* for when it is false. One of these transition is enabled when pc_L reaches n, the line number of the branch instruction. It sets the corresponding flag b_n and updates pc_L to the next statement to be executed. Note that for this transition the next label depends on the evaluated condition.

$$\begin{aligned} \mathit{BranchT}(\mathtt{n} : \mathtt{if(b)}) == \ & pc_L = n \land b = true \land \\ & pc'_L = next_L(n, true) \land \\ & b'_n = true \end{aligned}$$

$$\begin{aligned} \mathit{BranchF}(\mathtt{n} : \mathtt{if(b)}) == \ & pc_L = n \land b = false \land \\ & pc'_L = next_L(n, false) \land \\ & b'_n = false \end{aligned}$$

Additionally transitions are required in the store order graph to ensure the corresponding flow. Again there are two transitions. One of these must occur later than the transition for the branch statement in the load order graph. It is enabled when the head of pc_S has reached the label of the branch and pc_L

[7]We show the transitions for if statements here. Identical transitions exist for while statements.

has a value greater than the label (and hence the branch condition has been evaluated and the branch flag set).

$$\begin{aligned}
SBranchT(\texttt{n} : \texttt{if(b)}) == {} & head(pc_S) = n \wedge n <_L pc_L \ \wedge b_n = true \wedge \\
& pc'_S = (\textbf{if } next_S(n, true) \neq 0 \\
& \qquad \textbf{then } \langle next_S(n, true)\rangle \frown tail(pc_S) \\
& \qquad \textbf{else } tail(pc_S))
\end{aligned}$$

$$\begin{aligned}
SBranchF(\texttt{n} : \texttt{if(b)}) == {} & head(pc_S) = n \wedge n <_L pc_L \ \wedge b_n = false \wedge \\
& pc'_S = (\textbf{if } next_S(n, false) \neq 0 \\
& \qquad \textbf{then } \langle next_S(n, false)\rangle \frown tail(pc_S) \\
& \qquad \textbf{else } tail(pc_S))
\end{aligned}$$

5.5. Fences and RMWs

A fence at line n occurs when both pc_L and pc_S have reached n. The latter enforces that all *g-stores* before the fence have occurred (and hence all registers are empty). The transition updates pc_L and pc_S to their next values.

$$\begin{aligned}
Fence(\texttt{n} : \texttt{fence}) == {} & pc_L = n \wedge head(pc_S) = n \wedge \\
& pc'_L = next_L(pc_L) \wedge \\
& pc'_S = (\textbf{if } next_S(n) \neq 0 \\
& \qquad \textbf{then } \langle next_S(n)\rangle \frown tail(pc_S) \\
& \qquad \textbf{else } tail(pc_S))
\end{aligned}$$

An RMW combines a store with a fence.[8] Like a fence it requires that both pc_L and pc_S have reached its line number. It will set a variable v_a to a value when a certain condition b holds, and not change it otherwise. Hence, like branches, we model it with two transitions. In each a flag r_n is set accordingly which directs the *next* pointer.

$$\begin{aligned}
RMWT(\texttt{n} : \texttt{RMW}(\texttt{b}, \texttt{v}_\texttt{a}, \texttt{val})) == {} & pc_L = n \wedge head(pc_S) = n \wedge b = true \wedge \\
& r'_n = true \wedge \\
& v'_a = val \wedge \\
& pc'_L = next_L(pc_L) \wedge \\
& pc'_S = (\textbf{if } next_S(n) \neq 0 \\
& \qquad \textbf{then } \langle next_S(n)\rangle \frown tail(pc_S) \\
& \qquad \textbf{else } tail(pc_S))
\end{aligned}$$

$$\begin{aligned}
RMWF(\texttt{n} : \texttt{RMW}(\texttt{b}, \texttt{v}_\texttt{a}, \texttt{val})) == {} & pc_L = n \wedge head(pc_S) = n \wedge b = false \wedge \\
& r'_n = false \wedge \\
& pc'_L = next_L(pc_L) \wedge \\
& pc'_S = (\textbf{if } next_S(n) \neq 0 \\
& \qquad \textbf{then } \langle next_S(n)\rangle \frown tail(pc_S) \\
& \qquad \textbf{else } tail(pc_S))
\end{aligned}$$

[8]We assume the fence occurs whether the modification takes place or not.

6. The seqlock example on TSO

As an example, we show how seqlock running on TSO is modelled. Consider the `write` operation whose code is given again below.

```
 write(d1,d2) {
1   acquire;
2   c++;
3   x1 = d1;
4   x2 = d2;
5   c++;
6   release;
 }
```

Following our methodology, we first replace the statements `c++` and `acquire` with their equivalent sequences of atomic steps (`release` is already atomic; it is a store of 1 to the `lock` variable). `c++` becomes `local_c=c+1; c=local_c`. We implement `acquire` as `do {} while(RMW(lock=1,lock,0)=false)`, i.e., an RMW to evaluate the loop condition followed by a branch to either the beginning of the loop (if the condition is true) or to after the loop (if it is false).

Next we derive the orders $<_L$ and $<_S$ from the program text. $<_L$ is just the program order which, in this example, can be represented by the regular expression $(0, (1, 12)^*, 2, 22, 3, 4, 5, 52, 6, 0)$ where 12, 22 and 52 are additional lines due to the breaking the statements `acquire` and `c++` into atomic steps. $<_S$ is simply $<_L$ minus the line numbers of loads, i.e., $(0, (1, 12)^*, 22, 3, 4, 52, 6, 0)$. These orders are reflected in the pre and post values of pc_L and pc_S in the transitions below.

We define a transition for each line of code following the rules of Section 5. For example, $\mathit{Invoke}(\texttt{write(d1, d2)})$ gives rise to the following transition.

$$\begin{aligned}\mathit{Invoke}(\texttt{write}(\texttt{d1}, \texttt{d2})) == {} & pc_L = 0 \wedge pc'_L = 1 \wedge pc'_S = pc_S \frown \langle 1 \rangle \wedge \\ & in' = in \frown \langle (d1, d2) \rangle\end{aligned}$$

This models the invocation of the `write` operation. It appends the input values $d1$ and $d2$ to the input sequences in_1 and in_2 respectively.

The transitions for the RMW modelling the `acquire` command requires that both pc_L and the head of pc_S are 1 and updates both to their next positions in their order, setting the lock to 0 if it is successful.

$$\begin{aligned}&\mathit{RMWT}(\texttt{1} : \texttt{RMW}(\texttt{lock} = \texttt{1}, \texttt{lock}, \texttt{0})) == \\ &\qquad pc_L = 1 \wedge \mathit{head}(pc_S) = 1 \wedge \mathit{lock} = 1 \wedge \\ &\qquad r'_1 = \mathit{true} \wedge \mathit{lock}' = 0 \wedge pc'_L = 12 \wedge pc'_S = \langle 12 \rangle \frown \mathit{tail}(pc_S) \\ &\mathit{RMWF}(\texttt{1} : \texttt{RMW}(\texttt{lock} = \texttt{1}, \texttt{lock}, \texttt{0})) == \\ &\qquad pc_L = 1 \wedge \mathit{head}(pc_S) = 1 \wedge \neg\, \mathit{lock} = 1 \wedge \\ &\qquad r'_1 = \mathit{false} \wedge pc'_L = 12 \wedge pc'_S = \langle 12 \rangle \frown \mathit{tail}(pc_S)\end{aligned}$$

The branch instruction is represented by the pairs of *Branch* and *SBranch* transitions below.

$$\begin{array}{l} \mathit{BranchT}(\mathtt{12:while(RMW(lock=1,lock,0)=0)}) == \\ \qquad\qquad\qquad pc_L = 12 \wedge r_1 = \mathit{true} \wedge pc_L' = 2 \wedge b_{12}' = \mathit{true} \\ \mathit{BranchF}(\mathtt{12:while(RMW(lock=1,lock,0)=0)}) == \\ \qquad\qquad\qquad pc_L = 12 \wedge r_1 = \mathit{false} \wedge pc_L' = 1 \wedge b_{12}' = \mathit{false} \\ \mathit{SBranchT}(\mathtt{12:while(RMW(lock=1,lock,0)=0)}) == \\ \qquad\qquad\qquad \mathit{head}(pc_S) = 12 \wedge 12 <_L pc_L \wedge b_{12} = \mathit{true} \wedge \\ \qquad\qquad\qquad pc_S' = \langle 22 \rangle \frown \mathit{tail}(pc_S) \\ \mathit{SBranchF}(\mathtt{12:while(RMW(lock=1,lock,0)=0)}) == \\ \qquad\qquad\qquad \mathit{head}(pc_S) = 12 \wedge 12 <_L pc_L \wedge b_{12} = \mathit{false} \wedge \\ \qquad\qquad\qquad pc_S' = \langle 1 \rangle \frown \mathit{tail}(pc_S) \end{array}$$

The first step of the `c++` statements illustrates bypassing: it loads c's value from the global variable if the register r_c is empty, otherwise it loads the last value of the register. This value is incremented and stored into local variable $local_c$. For the statement at line 2, we have:

$$\begin{array}{l} \mathit{Load}(\mathtt{2:local\_c=c+1}) == \\ \qquad\qquad pc_L = 2 \wedge pc_L' = 22 \wedge \\ \qquad\qquad \mathit{local\_c}' = (\mathbf{if}\ r_c = \langle\,\rangle\ \mathbf{then}\ c+1\ \mathbf{else}\ \mathit{last}(r_c)+1) \end{array}$$

The second step of `c++` stores the value of `local_c` to the register associated with `c`. A *g-store* transition will later write the stored value to `c`. This transition requires that the *l-store* to the register has occurred and so checks that the register is not empty. For the `c++` statement at line 2, we have:

$$\begin{array}{l} \mathit{Store}(\mathtt{22:c=local\_c}) == pc_L = 22 \wedge pc_L' = 3 \wedge r_c' = r_c \frown \langle local_c \rangle \\ \mathit{GStore}(\mathtt{22:c=local\_c}) == \\ \qquad\qquad \mathit{head}(pc_S) = 22 \wedge r_c \neq \langle\,\rangle \wedge \\ \qquad\qquad c' = \mathit{head}(r_c) \wedge r_c' = \mathit{tail}(r_c) \wedge pc_S' = \langle 3 \rangle \frown \mathit{tail}(pc_S) \end{array}$$

The stores to `x1` and `x2` at lines 3 and 4, respectively, do not involve bypassing since they are storing values from local (input) variables. For the store at line 3 we have:

$$\mathit{Store}(\mathtt{3:x1=d1}) == pc_L = 3 \wedge pc_L' = 4 \wedge r_{x1}' = r_{x1} \frown \langle \mathit{last}(in).1 \rangle$$

where $t.1$ returns the first element of a tuple t.

The `release` statement is just a store of 1 to the `lock` variable. It sets pc_L to 0 as it is the last step in the `write` operation, and adds $\perp$ to the sequence of outputs *out* (to indicate no output).

$$\begin{array}{l} \mathit{Store}(\mathtt{6:lock=1}) == \\ \qquad\qquad pc_L = 6 \wedge pc_L' = 0 \wedge r_{lock}' = r_{lock} \frown \langle 1 \rangle \wedge out' = out \frown \langle \perp \rangle \end{array}$$

Since the `read` operation does not write to any global variable there are no store steps that may be delayed. The steps of the `read` operation are either loads with bypassing or branch instructions and are modelled similarly to those of the `write` operation.

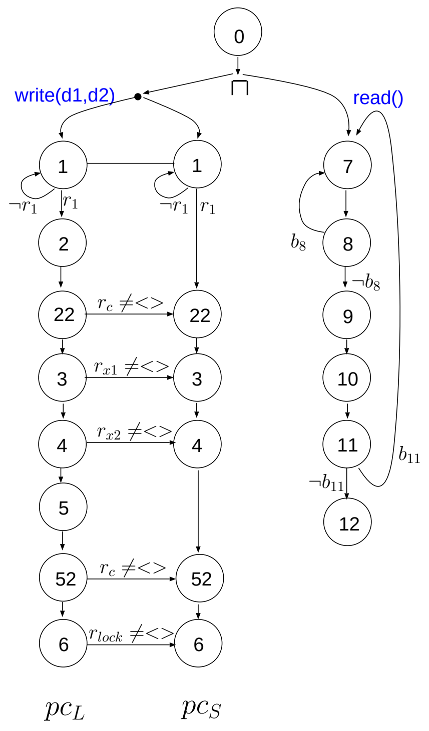

Figure 6: A flow graph of the seqlock under TSO

The control flow graph of the transition system is depicted in Figure 6. From the idle state (0) the system non-deterministically chooses which operation to invoke (depicted by ⊓). The `write` operation has a parallel flow along the load counter and the store counter. Since the `read` operation has no store instructions, a store counter is not required. The label of the RMW instruction (included in the `acquire` instruction on line 1) is shared between load and store counter. This is depicted by the connecting line between the corresponding nodes. Where branch conditions affect the flow the edges are labelled accordingly. For simplicity we omit the flow from the last label of the counters back to the idle state.

As a proof of concept, we encoded the transitions of the generated system for model checking. We adopted the approach of Smith [17] and chose linearization points based on the approach of [32]. Instead of TLC (the model checker used in [17]), we used the symbolic model checker NuSMV [19] as the tool handles the large assertions required for models on weak memory more efficiently. We restricted $c < 4$ and register sequences to length 2, which is sufficient for this example. Since only one step is performed and the check on the post-state is a simple invariant check, the model checking is very fast: for each of the checks (1) to (3) described in Section 2, the model checking returns within a few seconds on seqlock modelled in TSO. If a violation is encountered a two-state counterexample is generated which is easily analysed by the user providing the scenario that led to the violation. Check (4) of Section 2, the initialisation check, is also performed within a few seconds. The models are available at `http://staff.itee.uq.edu.au/kirsten/LinModels/SeqLock.html`.

On TSO, seqlock linearizes with respect to the specification in Figure 1 from which we deduce that it operates correctly.[9]

[9]Note that correctness in this case is in the context of an *operation-race free* client; see [32] for details.

7. A model for ARMv8

ARMv8 [3] is the latest version of the ARM multicore processor [4]. Unlike previous versions, it is multi-copy atomic, greatly simplifying the allowed behaviour of programs running on it. ARM is a significantly weaker memory model than TSO, allowing more reorderings, e.g., in TSO orders of writes on a single thread are always maintained, whereas on ARM they are only maintained when to the same variable or another dependency exists between them (as detailed in Section 7.5).

ARM also allows speculative execution to affect program behaviour. Speculative execution is when instructions in a branch are executed before the branch condition is evaluated. This is to optimise performance when evaluating the guard condition takes time. If the wrong branch is chosen, the results are discarded.

Additionally ARM supports more types of fences. As well as full fences, as in TSO, there are store fences and control fences. The latter can be used by the programmer to prevent speculative execution if needed.

A number of formal models of ARM exist, both of previous versions [33, 4, 20] and of the latest version, ARMv8 [3, 21]. We adopt the operational semantics of ARMv8 provided by Colvin and Smith [21]. This semantics clearly identifies allowable reorderings of instructions including reorderings with branch instructions which result in speculative execution, and has been validated against approximately 10,000 existing litmus tests run on ARM hardware.

7.1. Transition System Model

To construct the transition system of a concurrent object under ARMv8, we need to first parse the object's code recording dependencies between lines of code which prevent reorderings of instructions where necessary. These dependencies are detailed in the following sections. They are used to generate a load and store order, enforced by pc_a^L and pc_a^S respectively, for each global variable a since the order between loads and stores is only maintained for the same address. In addition, for dependencies on control fences, branches[10] and *address shifting* (see Section 7.8) which are not captured by these orders, we introduce flags. If such a dependency requires a line m to occur before a line n which appears later in the program order than m, we have a flag that is set by m and which guards the occurrence of n.

For each order we have a sequence of program counters, one for each overlapping operation (for further details see Section 7.3). That is, for address a we have a sequence of program counters for loads, $pc_a^L : \text{seq}\,\mathcal{L}^L(a)$, and a sequence of program counters for stores, $pc_a^S : \text{seq}\,\mathcal{L}^S(a)$. The set of labels $\mathcal{L}^L(a)$ includes all line numbers of loads and stores associated with a, branches whose conditions refer to a, and all full fences and RMWs. $\mathcal{L}^S(a)$ denotes the set of

[10]Branches that do not refer to global variables do not appear in any load or store order.

labels of stores associated with a, store fences, and all full fences and RMWs (to prevent reordering of store fences with these instructions).

Since speculative execution is possible, we need to check transitions corresponding to all branches. To enable this, we set all branch conditions for an operation nondeterministically on invocation. This ensures all variables follow the same flow of control.

In the following we denote with v_a the global variable at address a, v_l a local variable, b_n the flag which indicates which branch the code is to follow after the branch instruction at line n, and $f_{n,m}$ the flag that is set by line n which guards the occurrence of line m. We refer to the program order with $<_P$ and denote the load order for every global address a with $<_a^L = \langle \mathcal{L}^L(a), next^L(a) \rangle$ and its store order as $<_a^S = \langle \mathcal{L}^S(a), next^S(a) \rangle$.

Initially, $pc_a^L = pc_a^S = \langle \, \rangle$ for all a and all flags are false. The transitions are defined in the following sections.

7.2. Dependencies

The semantics of ARMv8 in [20, 21] prescribes when two instructions cannot be reordered under the ARM semantics. These specified constraints impose dependencies between instructions which need to be checked in the corresponding transitions.

Each load and store order of an address enforces naturally the prescribed order of instructions (referring to the same address) by means of the *next* pointer which traverses through these orders.

For dependent instructions which do not occur in such an order, we use two mechanisms. When the label of the earlier instruction appears in the load counter of a different address a, we check that a's load counter has passed the line m at which the instruction occurs. We denote the set of pairs of all such addresses a and lines m as $before(n)$, indicating that the corresponding instructions must occur *before* line n.

When the earlier instruction's label is not part of any load counter (e.g., when it is a control fence, a branch without a direct reference to a global variable, or when it involves address shifting), we check that a flag associated with the instruction has been set. We denote the set of all labels of instructions which set such flags for the instruction at n as $flags(n)$. The flag $f_{m,n}$ ensures that the earlier instruction at m occurs before the instruction at n, and thus their reordering is prohibited.

The required check on both ordering-enforcing mechanisms is encoded in $ready(n)$ which also resets the flags back to false.

$$\begin{aligned} ready(n) == {} & (\forall (a,m) : before(n) \cdot m <_P head\, pc_a^L) \wedge \\ & (\forall\, m : flags(n) \cdot f_{m,n} = true \wedge f'_{m,n} = false) \end{aligned}$$

This check is used in the transitions described below.

7.3. Overlapping of operations

Since the ARMv8 memory model is significantly weaker than the TSO memory model, more overlapping between consecutively invoked operations may occur. Overlapping operations can occur when an operation call returns before all instructions of that operation are completed. A second operation can then be invoked and may execute its (independent) instructions. For example, operation `write` of seqlock might return (i.e., release the lock) before variable `x2` has been updated, as the instructions are independent of each other and reordering is permitted.

When two different operations (e.g., operations `write()` and `read()`) overlap our model easily handles the reordering of instructions making use of the sequences of counters. When we have overlapping instances of the same operation, we need to create a separate name space for each instance (as happens on the processor level). This can be achieved by duplicating an operation and giving the instructions of the copy fresh labels (that differ from the original labels) as well as using fresh names for local variables and input variables (as introduced in the next subsection).

7.4. Invocations and returns

Since we do not have a counter following program order such as pc_L in TSO, we use an additional order $<_{ret}$ and program counter, pc_{ret}, to keep track of the instructions which must occur before an operation returns.

The invocation of an operation `op` is enabled when $pc_{ret} = 0$ (i.e., any prior operation has returned). It updates pc_{ret} to the first line number for ret in the invoked operation. Similarly, it updates pc_a for each a. It nondeterministically chooses which branches (occurring at line numbers $m_1, ..., m_k$) to take in the operation. Input values are stored in a variable in_{op}.

$$\begin{aligned}
Invoke(\mathtt{op}(\mathtt{val_1}, \ldots, \mathtt{val_n})) == {} & pc_{ret} = 0 \wedge pc'_{ret} = next_{ret}(0) \wedge \\
& (\forall\, a : Addr \cdot {pc^L_a}' = pc^L_a \frown \langle next^L_a(0)\rangle \wedge \\
& \qquad\qquad {pc^S_a}' = pc^S_a \frown \langle next^S_a(0)\rangle) \wedge \\
& (\forall\, i : 1..k \cdot b'_{m_i} \in \{true, false\}) \wedge \\
& {in_{op}}' = (val_1, \ldots, val_n)
\end{aligned}$$

A return statement at line n is treated like an assignment of the return value to a local variable. Under ARM, such an assignment can only occur after any fences and RMWs in the operation. It may also require other instructions appearing earlier in the program text to have occurred (see rules in Sections 7.5 and 7.6). These dependencies are covered through predicate $ready()$. The return of an operation `op` at line n is enabled when $ready(n)$ holds and $pc_{ret} = n$. It sets pc_{ret} back to 0 and updates a variables out_{op} with the outputs.

$$\begin{aligned}
Return(\mathtt{n} : \mathtt{return}(\mathtt{val_1}, \ldots, \mathtt{val_n})) == {} & ready(n) \wedge \\
& pc_{ret} = n \wedge pc'_{ret} = 0 \wedge \\
& {out_{op}}' = (val_1, \ldots, val_n)
\end{aligned}$$

When there is no explicit return statement (as in the write operation of seqlock), the final statement of the operation sets pc_{ret} to 0 and updates out_{op} to $\perp$, indicating no output.

7.5. Loads and stores

In ARM, reordering of loads and stores are governed by four constraints formalised in [20, 21]. These constraints ensure that the sequential semantics of the thread on which the reordering occurs is unchanged. In fact, they are common to all contemporary weak memory models. An assignment $v := e$ can be reordered with an assignment $u := f$ if

1. v and u are distinct variables,
2. v is not free in f,
3. u is not free in e, and
4. e and f do not reference any common global variables.

In practice, case 2 may be circumvented by *forwarding*. This refers to taking into account the effect of the assignment moved later on the expression of the other assignment. For example, the code $x = e;\ y = x$ where e does not reference global variables can be reordered to $y = e;\ x = e$. To allow this in our model, when a variable x is assigned a value e not involving global variables, we change each subsequent reference to x in any expression with the value e.

To construct the transitions corresponding to a load or store occurring at line n, we first parse the program text to perform these forwarding replacements. Any dependencies that are not captured by the load and store orders are captured by $ready(n)$.

The transition for a load instruction from an address a at line n is enabled when $ready(n)$ holds and the first load counter in pc_a^L is n. It updates the counter to the next value in $<_a^L$ or removes it when the counter returns to 0.

$$\begin{aligned}
Load(\mathtt{n} : \mathtt{v_l} = \mathtt{e}(\mathtt{v_a})) == {} & ready(n) \wedge \\
& head\ pc_a^L = n \wedge \\
& pc_a^{L\prime} = (\textbf{if } next_a^L(n) \neq 0 \\
& \qquad \textbf{then } \langle next_a^L(n) \rangle \frown tail\ pc_a^L \\
& \qquad \textbf{else } tail\ pc_a^L) \wedge \\
& v_l' = e(v_a)
\end{aligned}$$

A store to address a at line n is enabled when $ready(n)$ holds and both the store counter pc_a^S and the load counter pc_a^L equal n (including loads into the store counter ensures that a store to a does not get reordered with an earlier load to a). The counters are updated to their next labels.

$$
\begin{aligned}
\mathit{Store}(\mathtt{n} : \mathtt{v_a} = \mathtt{val}) == {} & \mathit{ready}(n) \wedge \\
& \mathit{head}\, pc_a^S = n \wedge \mathit{head}\, pc_a^L = n \wedge \\
& pc_a^{S\prime} = (\textbf{if } \mathit{next}_a^S(n) \neq 0 \\
& \qquad \textbf{then } \langle \mathit{next}_a^S(n) \rangle \frown \mathit{tail}\, pc_a^S \\
& \qquad \textbf{else } \mathit{tail}\, pc_a^S) \wedge \\
& pc_a^{L\prime} = (\textbf{if } \mathit{next}_a^L(n) \neq 0 \\
& \qquad \textbf{then } \langle \mathit{next}_a^L(n) \rangle \frown \mathit{tail}\, pc_a^L \\
& \qquad \textbf{else } \mathit{tail}\, pc_a^L) \wedge \\
& v_a' = \mathit{val}
\end{aligned}
$$

7.6. Branches

The constraints on reordering branch instructions with other instructions in ARM are governed by the following rules formalised in [20, 21].

1. A load or store $v := e$ preceding a branch instruction with branching condition b can be reordered with the branch instruction if, and only if, v does not appear in b, and no global variables in e appear in b.
2. A load $v := e$ following a branch instruction with branching condition b can be reordered with the branch instruction if, and only if, no global variables in e appear in b.
3. A store $v := e$ following a branch instruction can never be reordered with it.
4. A branch instruction with branching condition b_1 can be reordered with another branch instruction with branching condition b_2 if, and only if, b_1 and b_2 do not have any global variables in common.

Case 1 captures branches being evaluated early. There are two situations to consider. Firstly, if a branch condition refers to an address a then the branch cannot be reordered before any load or store to a. This is captured by including the branch in the load order for a. In the rule below, we let the set of all addresses referred to by branch condition b be denoted by $\mathit{addr}(b)$.

Secondly, if a branch condition does not refer to address a but refers to a local variable which loads from address a then the branch cannot be reordered before this load. This is captured by ensuring the load order of a has passed the line m on which the load occurs. In the rule below, this constraint is captured in $\mathit{ready}(n)$.

Case 2 corresponds to speculative execution. If the load is from an address a referred to in the branch condition then it should not be reordered before the branch. This is captured by including the branch in the load order for a.

Case 3 prevents speculative execution of stores. This is necessary since if it is later determined that the branch should not be executed, it is necessary to discard all results. This cannot be done with stores which other threads may have seen. To capture this constraint, the branch sets flags for the next store in each load order. (We deal with the case where there is no next store in the load order after the rule below.) Note that only the label to the next store per address needs to have a flag as subsequent stores are restricted through the load and store orders. In the rule below, we denote the set of labels of such stores as $\mathit{after}(n)$, indicating that they must occur after line n.

Case 4 concerns reordering of two branch instructions. This follows naturally as both branch labels would be included in the load orders of common addresses.

$$
\begin{aligned}
\mathit{Branch}(\mathtt{n} : \mathtt{if(b)}) == {} & \mathit{ready}(n) \wedge \\
& (\forall\, a : \mathit{addr}(b) \cdot \mathit{head}\, pc_a^L = n \wedge \\
& \qquad {pc_a^L}' = (\textbf{if } \mathit{next}_a^L(n) \neq 0 \\
& \qquad\qquad \textbf{then } \langle \mathit{next}_a^L(n) \rangle \frown \mathit{tail}\, pc_a^L \\
& \qquad\qquad \textbf{else } \mathit{tail}\, pc_a^L)) \wedge \\
& (\forall\, m : \mathit{after}(n) \cdot f'_{n,m} = \mathit{true})
\end{aligned}
$$

If there is no store instruction to an address a after a branch then the branch should still guard the occurrence of any stores to that address in any operation call that follows. To cover this case we add a "dummy" label to the end of the load order of that address. This label will be in the set $\mathit{after}(n)$ of the rule above.

The "skip" transition at that label is enabled when all the flags for it have been set. It resets the flags to false and allows the load counter for a to move on to the next operation. As previously, we denote the set of all labels of instructions which set such flags as $\mathit{flags}(n)$.

$$
\begin{aligned}
\mathit{Skip}(\mathtt{n} : \mathtt{skip(a)}) == {} & \mathit{head}\, pc_a^L = n \wedge \\
& (\forall\, m : \mathit{flags}(n) \cdot f_{m,n} = \mathit{true} \wedge f'_{m,n} = \mathit{false}) \wedge \\
& {pc_a^L}' = \mathit{tail}\, pc_a^L
\end{aligned}
$$

7.7. Fences and RMWs

A full fence (DMB or DSB in ARM) at line n is enabled when all previous instructions have occurred, i.e., when all program counters have reached line n. All program counters are updated to their next value.

$$
\begin{aligned}
\mathit{Fence}(\mathtt{n} : \mathtt{fence}) == {} & (\forall\, a : \mathit{Addr} \cdot \mathit{head}\, pc_a^L = n \wedge \mathit{head}\, pc_a^S = n \wedge \\
& \qquad {pc_a^S}' = (\textbf{if } \mathit{next}_a^S(n) \neq 0 \\
& \qquad\qquad \textbf{then } \langle \mathit{next}_a^S(n) \rangle \frown \mathit{tail}\, pc_a^S \\
& \qquad\qquad \textbf{else } \mathit{tail}\, pc_a^S) \wedge \\
& \qquad {pc_a^L}' = (\textbf{if } \mathit{next}_a^L(n) \neq 0 \\
& \qquad\qquad \textbf{then } \langle \mathit{next}_a^L(n) \rangle \frown \mathit{tail}\, pc_a^L \\
& \qquad\qquad \textbf{else } \mathit{tail}\, pc_a^L))
\end{aligned}
$$

An RMW at line n has a full fence and hence is enabled when all previous instructions have occurred. As in TSO, there is a transition for each evaluation of the RMW's condition b which updates a flag r_n which directs the flow to the succeeding instructions (through next). If condition b is true variable v_a gets updated.

$$
\begin{array}{l}
\mathit{RMWT}(\mathtt{n} : \mathtt{RMW}(\mathtt{b}, \mathtt{v_a}, \mathtt{val})) == (\forall\, a : \mathit{Addr} \cdot \mathit{head}\, \mathit{pc}_a^L = n \wedge \mathit{head}\, \mathit{pc}_a^S = n \wedge \\
\quad b = \mathit{true} \wedge \\
\quad r_n' = \mathit{true} \wedge \\
\quad v_a' = \mathit{val} \wedge \\
\quad \mathit{pc}_a^{S\prime} = (\textbf{if } \mathit{next}_a^S(n) \neq 0 \\
\qquad \textbf{then } \langle \mathit{next}_a^S(n) \rangle \frown \mathit{tail}\, \mathit{pc}_a^S \\
\qquad \textbf{else } \mathit{tail}\, \mathit{pc}_a^S) \wedge \\
\quad \mathit{pc}_a^{L\prime} = (\textbf{if } \mathit{next}_a^L(n) \neq 0 \\
\qquad \textbf{then } \langle \mathit{next}_a^L(n) \rangle \frown \mathit{tail}\, \mathit{pc}_a^L \\
\qquad \textbf{else } \mathit{tail}\, \mathit{pc}_a^L))
\end{array}
$$

$$
\begin{array}{l}
\mathit{RMWF}(\mathtt{n} : \mathtt{RMW}(\mathtt{b}, \mathtt{v_a}, \mathtt{val})) == (\forall\, a : \mathit{Addr} \cdot \mathit{head}\, \mathit{pc}_a^L = n \wedge \mathit{head}\, \mathit{pc}_a^S = n \wedge \\
\quad b = \mathit{false} \wedge \\
\quad r_n' = \mathit{false} \wedge \\
\quad \mathit{pc}_a^{S\prime} = (\textbf{if } \mathit{next}_a^S(n) \neq 0 \\
\qquad \textbf{then } \langle \mathit{next}_a^S(n) \rangle \frown \mathit{tail}\, \mathit{pc}_a^S \\
\qquad \textbf{else } \mathit{tail}\, \mathit{pc}_a^S) \wedge \\
\quad \mathit{pc}_a^{L\prime} = (\textbf{if } \mathit{next}_a^L(n) \neq 0 \\
\qquad \textbf{then } \langle \mathit{next}_a^L(n) \rangle \frown \mathit{tail}\, \mathit{pc}_a^L \\
\qquad \textbf{else } \mathit{tail}\, \mathit{pc}_a^L))
\end{array}
$$

A store fence (DMB.ST or DSB.ST in ARM) at line n is enabled when all previous stores have occurred. Since store fences are contained in the store orders of all variables, the occurrence of a store fence is enabled only when all previous stores have occurred. It also acts as a guard on all succeeding stores. Upon occurrence of the store fence the store counters for all addresses get set to their next value.

$$
\begin{array}{l}
\mathit{StoreFence}(\mathtt{n} : \mathtt{fence.st}) == (\forall\, a : \mathit{Addr} \cdot \mathit{head}\, \mathit{pc}_a^S = n \wedge \\
\quad \mathit{pc}_a^{S\prime} = (\textbf{if } \mathit{next}_a^S(n) \neq 0 \\
\qquad \textbf{then } \langle \mathit{next}_a^S(n) \rangle \frown \mathit{tail}\, \mathit{pc}_a^S \\
\qquad \textbf{else } \mathit{tail}\, \mathit{pc}_a^S)))
\end{array}
$$

The final type of fence is a control fence (ISB in ARM). A control fence can be placed between a branch instruction and following loads to prevent the loads being speculatively executed. That is, a branch before a control fence cannot be reordered with it, and a load after a control fence cannot be reordered with it.

The dependencies with earlier branches can be captured by $\mathit{ready}(n)$, using flags if the branches do not appear in any load order. The dependencies with later loads can be captured by setting flags. Note that only the next load per address (after the control fence) is required to have a flag set. (When there is no such next load we use a "dummy" label and a "skip" transition as in

Section 7.6.) In the rule below, we denote the set of labels of such loads as *after*(n).

$$\mathit{ControlFence}(\mathtt{n} : \mathtt{cfence}) == \mathit{ready}(n) \wedge$$
$$(\forall\, m : \mathit{after}(n) \cdot f'_{m,n} = \mathit{true})$$

7.8. Address shifting

The address a an instruction loads from, or stores to, in ARM may be shifted by a number of bytes. We denote this by $\mathit{shift}(a, e)$ where e is an expression evaluating to the number of bytes. This creates an additional constraint on reordering loads and stores not already covered in Section 7.5. Stores occurring after the instruction using address shifting should not be reordered with it [4]. This is in case the shift amount results in an invalid address causing an exception to be thrown. In such cases, the stores should not be visible to other threads. This is captured by allowing the load or store to set flags for the next store in each load order, or by adding a "dummy" label and "skip" operation when there is no such load. The details are as for branches in Section 7.6.

7.9. Load speculation

A further aspect of address shifting is that in some circumstances a load $r_2 = a$ (at line n) may be reordered before a load $r_1 = \mathit{shift}(a, e)$ (at an earlier line m), even though both loads refer to a. This reordering is allowed whenever the load into r_2 does not load a value of a that was written before the value read by the load into r_1. This optimisation has no observable effect (since stores following the load to r_2 cannot be reordered before the load to $r1$ as described above). Hence, we do not need to support this in our model.

7.10. Write elimination

An additional aspect of ARM processors is that when there are consecutive stores to a variable x on a thread, the first write can effectively be eliminated: globally it is always a valid behaviour that another thread did not see the effect of the first write because the second could have occurred immediately after it.

For example, given a program such as

$$\begin{array}{l}\mathtt{1 : y = 0}\\ \vdots\\ \mathtt{m : x = y}\\ \vdots\\ \mathtt{n : x = 0}\end{array}$$

If the write at line n can be reordered before all instructions at lines $m+1...n-1$ then the write at line m can be eliminated. Hence, the write at line n may be reordered even earlier including before the write to y at line 1 which would not otherwise have been possible due to the dependency between lines 1 and m.

```
write(d1,d2) {
       do {}
   1     while (RMW(lock=1,lock,0)=false);
   2   local_c = c+1;
   22  c = local_c;
   23  fence.st;
   3   x1 = d1;
   4   x2 = d2;
   5   local_c = c+1;
   52  c = local_c;
   6   lock = 1;
       }
```

```
 read() {
      word c0;
      do {
        do {
7          c0 = c;
8       } while(c0%2!=0);
82      cfence;
9       d1 = x1;
10      d2 = x2;
102     fence
11    } while(c != c0);
12    return(d1,d2);
   }
```

Figure 7: Modified seqlock implementation

To capture this we create a second transition system which is identical to the first except that it leaves m out of the program counters for x. That is, if $m = next_x^L(k)$ and $n = next_x^L(m)$ we alter $next_x^L$ pointwise so that $n = next_x^L(k)$ and otherwise $next_x^L$ is unchanged. The same changes are imposed on the next function of the store order, $next_x^S$. Both transition systems are checked to ensure both possible behaviours (i.e., when the write is eliminated or not) are covered.

8. The seqlock example on ARMv8

The example implementation from Figure 2 is modified for illustration purposes in Figure 7. Some instructions are replaced following the implementation outlined in Section 6, and a `fence`, control fence (`cfence`), and a store fence (`fence.st`) are added to the code.[11] The program order for the modified code is as follows, where 0 denotes the idle state.

$$<_{write} = \langle 0, 1, 2, 22, 23, 3, 4, 5, 52, 6, 0\rangle \qquad <_{read} = \langle 0, 7, 8, 82, 9, 10, 102, 11, 12, 0\rangle$$

The counters for `write` are given as follows where the label of the RMW is added to all load counters

$$pc_{lock}^L = \langle 0, 1, 6, 0\rangle \quad pc_c^L = \langle 0, 1, 2, 22, 5, 52, 0\rangle \quad pc_{x1}^L = \langle 0, 1, 3, 0\rangle$$
$$pc_{x2}^L = \langle 0, 1, 4, 0\rangle$$

and the label of the the RMW and the store fence 23 is added to all store counters

$$pc_{lock}^S = \langle 0, 1, 23, 6, 0\rangle \quad pc_c^S = \langle 0, 1, 22, 23, 52, 0\rangle \quad pc_{x1}^S = \langle 0, 1, 23, 3, 0\rangle$$
$$pc_{x2}^S = \langle 0, 1, 23, 4, 0\rangle.$$

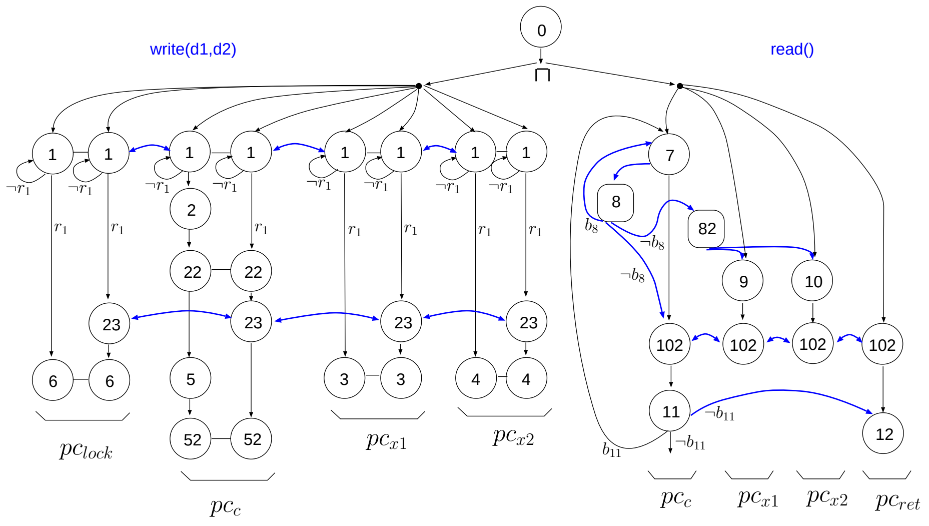

Figure 8: A concurrent flow graph of the modified seqlock implementation

The counters for `read` can be given similarly.

The control flow graph (on a single thread) of the modified `seqLock` example is depicted in Figure 8. The flow graph includes concurrent flows along the different address counters, as well as conditional flow where two outgoing edges are labelled with guards (e.g., r_1 and $\neg r_1$). From the idle state 0 a choice is made (depicted by the $\sqcap$ symbol) of whether to call a write or a read operation. The arrows indicate the flows along the program counters for each address (modelled by the *next* functions that direct the load and store counters), or depict cross dependencies between program counters, (e.g. the branch instruction at line 8 is guarded by the load instruction at line 7, and the control fence at line 82 guards the two load instructions at line 9 and 10).[12] Labels that are present in two counters are linked in the graph as both counters are checked before proceeding with the flow. (For readability we omit the edges pointing from the last instruction of each counter back to the idle state 0, and depict the outgoing edges on node 102 only once. Where there are no store instructions the store counters are omitted, e.g., for `read`.)

8.1. Model checking

The rules in Section 7 allows us to construct a transition system for a concurrent object running on ARMv8 which can then be used to verify that the object is linearizable with respect to its specification. Note that speculatively executing branches does not lead to erroneous model checking results. In particular, it does not lead to a proof obligation failing when it should not. To see

[11]Note that these additions are merely to demonstrate how various language features are handled.

[12]The latter appear blue in the electronic version of this paper.

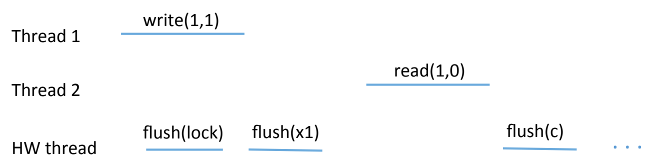

Figure 9: Linearizability fails on XC

why this is the case, consider the following program.

$\texttt{if}(\texttt{y} > 0)$ `then` $\texttt{x} = 1$ `else` $\texttt{x} = 0$

Assume we have an invariant $y > 0 \Leftrightarrow x > 0$. In the case when $y = 0$ but the `then` branch is speculatively executed this invariant will not hold. This does not cause a problem for our proof method since the assertion before the `if` statement (see Section 2) will include $y = 0$ and the assertion at the start of the `then` branch will include both $y = 0$ and $y > 0$. Since this assertion evaluates to false, proof obligations (1) to (3) of Section 2 are trivially satisfied for the step x=1.

A NuSMV encoding of the seqlock implementation of Figure 2 using a subset of the ARMv8 rules (those needed for the similar weak memory model XC [2]) shows that it fails to linearize due to the liberal reordering allowing `x1` and `x2` to be set before `c` becomes odd, and after it becomes even. A counterexample generated by NuSMV relates to the scenario visualised in Figure 9. The write by Thread1 to `x1` has been flushed before its write to `c` allowing Thread2 to read the new value of `x1` together with the old value of `x2` (assumed to be 0).

To avoid this and similar counter-examples, fences are required before lines 3 and 5. Also, to ensure another thread does not start a write while `c` is odd, a fence is also required after line 5 (so that `release` cannot happen before the final increment of `c`). Finally, fences are required before line 9 and after line 10 in the `read` operation, to ensure values are read into `d1` and `d2` only when `c0` is even, and the final reads occur before `c` changes value. The models are available at `http://staff.itee.uq.edu.au/kirsten/LinModels/SeqLock.html`.

9. Conclusions

This paper has presented approaches for generating transition systems for concurrent objects running on the TSO and ARMv8 weak memory models. Modelling the behaviour of programs under hardware weak memory models has also been investigated by others, e.g., [34, 31, 35, 36, 30, 37, 38].

While some of this other work presents very similar transition systems (in particular the two control flow graphs and the split of stores into two steps is similar in [30, 31]), none of the approaches uses the notion of linearizability to verify correctness. Linearizability, however, has the advantage over other correctness conditions that it has a proof method that is not only thread-local but also step-local. Although the user of such a method has to provide assertions (see Section 2) that enable the step-local proof, it provides the benefit of having a proof for an arbitrary number of threads and arbitrary sequences of operation calls.

Although the focus of this paper was on generating transition systems, as a proof of concept we also related our experience with using a generated transition system with the NuSMV model checker. This required manually providing the assertions which hold after each program step: a non-trivial task on a weak memory model. These assertions can be generated for concurrent objects running on sequentially consistent architectures [26] and it is future work to extend this approach to objects under weak memory models. In [38] the assertion-based Owicki-Gries proof system is extended to achieve soundness of the calculus for weak memory models. A similar approach will drive our further investigation.

Acknowledgement This work was supported by Australian Research Council Discovery Grant DP160102457, and by the UK EPSRC grant EP/R032351/1.

References

[1] P. Sewell, S. Sarkar, S. Owens, F. Nardelli, M. Myreen, x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors, Commun. ACM 53 (7) (2010) 89–97.

[2] D. Sorin, M. Hill, D. Wood, A Primer on Memory Consistency and Cache Coherence, Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2011.

[3] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, P. Sewell, Simplifying ARM concurrency : multicopy-atomic axiomatic and operational models for ARMv8, in: POPL 2018, ACM, 2018, pp. 19:1–19:29.

[4] S. Flur, K. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, P. Sewell, Modelling the ARMv8 architecture, operationally: Concurrency and ISA, in: POPL 2016, ACM, 2016, pp. 608–621.

[5] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, D. Williams, Understanding POWER multiprocessors, SIGPLAN Not. 46 (6) (2011) 175–186.

[6] M. Moir, N. Shavit, Concurrent data structures, Handbook of Data Structures and Applications (2004) 47:1–47:30.

[7] M. Herlihy, J. Wing, Linearizability: A correctness condition for concurrent objects, ACM Trans. Prog. Lang. Syst. 12 (3) (1990) 463–492.

[8] D. Amit, N. Rinetzky, T. Reps, M. Sagiv, E. Yahav, Comparison under abstraction for verifying linearizability, in: CAV 2007, Vol. 4590 of LNCS, Springer, 2007, pp. 477–490.

[9] C. Calcagno, M. Parkinson, V. Vafeiadis, Modular safety checking for fine-grained concurrency, in: SAS 2007, Vol. 4634 of LNCS, Springer, 2007, pp. 233–238.

[10] V. Vafeiadis, Modular fine-grained concurrency verification, Ph.D. thesis, University of Cambridge (2007).

[11] S. Doherty, L. Groves, V. Luchangco, M. Moir, Formal verification of a practical lock-free queue algorithm, in: FORTE 2004, Vol. 3235 of LNCS, Springer, 2004, pp. 97–114.

[12] J. Derrick, G. Schellhorn, H. Wehrheim, Proving linearizability via non-atomic refinement, in: IFM 2007, Vol. 4591 of LNCS, Springer, 2007, pp. 195–214.

[13] J. Derrick, G. Schellhorn, H. Wehrheim, Mechanically verified proof obligations for linearizability, ACM Trans. Program. Lang. Syst. 33 (1) (2011) 4:1–4:43.

[14] J. Derrick, G. Schellhorn, H. Wehrheim, Verifying linearisabilty with potential linearisation points, in: FM 2011, Vol. 6664 of LNCS, Springer, 2011, pp. 323–337.

[15] G. Schellhorn, H. Wehrheim, J. Derrick, A sound and complete proof technique for linearizability of concurrent data structures, ACM Trans. on Computational Logic 15 (4) (2014) 31:1–31:37.

[16] W. Reif, G. Schellhorn, K. Stenzel, M. Balser, Structured specifications and interactive proofs with KIV, in: Automated Deduction—A Basis for Applications, Vol. II, Kluwer, 1998, Ch. 1: Interactive Theorem Proving, pp. 13 – 39.

[17] G. Smith, Model checking simulation rules for linearizability, in: SEFM 2016, Vol. 9763 of LNCS, Springer, 2016, pp. 188–203.

[18] D. Bovet, M. Cesati, Understanding the Linux Kernel, 3rd Edition, O'Reilly, 2005.

[19] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking, in: E. Brinksma, K. G. Larsen (Eds.), CAV 2002, Vol. 2404 of LNCS, Springer, 2002, pp. 359–364.

[20] R. Colvin, G. Smith, A wide-spectrum language for verification of programs on weak memory models, in: K. Havelund, J. Peleska, B. Roscoe, E. de Vink (Eds.), FM 2018, Vol. 10951 of LNCS, Springer, 2018, pp. 240–257.

[21] R. Colvin, G. Smith, A high-level operational semantics for hardware weak memory models, CoRR abs/1812.00996.

[22] K. Winter, G. Smith, J. Derrick, Observational models for linearizability checking on weak memory models, in: 12th International Symposium on Theoretical Aspects of Software Engineering (TASE 2018), IEEE Computer Society Press, 2018, pp. 100–107.

[23] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann, 2008.

[24] S. Burckhardt, A. Gotsman, M. Musuvathi, H. Yang, Concurrent library correctness on the TSO memory model, in: ESOP 2012, Vol. 7211 of LNCS, Springer, 2012, pp. 87–107.

[25] S. Owicki, D. Gries, An axiomatic proof technique for parallel programs I, Acta Informatica 6 (4) (1976) 319–340.

[26] G. Smith, J. Derrick, Invariant generation for linearizability proofs, in: SAC 2016, ACM, 2016, pp. 1694–1699.

[27] O. Travkin, A. Mütze, H. Wehrheim, SPIN as a linearizability checker under weak memory models, in: HVC 2013, Vol. 8244 of LNCS, Springer, 2013, pp. 311–326.

[28] J. Derrick, G. Smith, B. Dongol, Verifying linearizability on TSO architectures, in: iFM 2014, Vol. 8739 of LNCS, Springer, 2014, pp. 341–356.

[29] J. Derrick, G. Smith, A framework for correctness criteria on weak memory models, in: FM 2015, Vol. 9109 of LNCS, Springer, 2015, pp. 178–194.

[30] O. Travkin, H. Wehrheim, Verification of concurrent programs on weak memory models, in: ICTAC 2016, Springer, 2016, pp. 3–24.

[31] T. Abe, T. Maeda, Concurrent program logic for relaxed memory consistency models with dependencies across loop iterations, Journal of Information Processing 25 (2017) 244–255.

[32] S. Doherty, J. Derrick, Linearizability and causality, in: Software Engineering and Formal Methods - 14th International Conference, (SEFM 2016), Vol. 9763 of LNCS, Springer, 2016, pp. 45–60.

[33] J. Alglave, L. Maranget, M. Tautschnig, Herding cats: Modelling, simulation, testing, and data mining for weak memory, ACM Trans. Program. Lang. Syst. 36 (2) (2014) 7:1–7:74.

[34] V. Still, J. Barnat, Model checking of C++ programs under the x86-tso memory model, in: J. Sun, M. Sun (Eds.), ICFEM 2018, Vol. 11232 of Lecture Notes in Computer Science, Springer, 2018, pp. 124–140.

[35] P. Abdulla, M. Atig, A. Bouajjani, T. Ngo, Context-bounded analysis for POWER, in: TACAS 2017, Vol. 10206 of LNCS, Springer, 2017, pp. 56–74.

[36] T. Abe, T. Maeda, A general model checking framework for various memory consistency models, International Journal on Software Tools for Technology Transfer 19 (5) (2017) 623–647.

[37] P. Abdulla, M. Atig, A. Bouajjani, T. Ngo, A load-buffer semantics for total store ordering, Logical Methods in Computer Science 14 (1:9) (2018) 1–46.

[38] O. Lahav, V. Vafeiadis, Owicki-Gries reasoning for weak memory models, in: ICALP 2015, Springer, 2015, pp. 311–323.