Observational models for linearizability checking on weak memory models

Kirsten Winter and Graeme Smith
School of Information Technology and Electrical Engineering, The University of Queensland, Australia
John Derrick
Department of Computing, University of Sheffield, UK

***Abstract*—Weak memory models 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 programs running on weak memory models since the variations in the executions need to be considered.**

Many approaches of modelling weak memory behaviour focus on *architectural models* to capture aspects of the hardware's architecture. In this paper, we investigate *observational models* of weak memory model behaviour which abstract from the underlying hardware architecture, and are instead derived from instruction reordering rules. This enables existing proof methods and tool support for linearizability to be reused. Specifically, we show how one existing proof method and associated model checking approach can be used to reason about programs running on the TSO and XC weak memory models.

I. Introduction

Linearizability [19] is the standard correctness condition for *concurrent objects*, i.e., objects that can be accessed simultaneously by more than one thread. A number of approaches have been developed for proving linearizability along with associated tool support [6], [8], [11]–[13], [17], [23], [30]. These approaches assume that the concurrent objects are running on a *sequentially consistent* architecture, i.e., one where instructions, e.g., loads to and stores from shared memory, take effect in the order they appear in the program. However, modern multicore architectures are not sequentially consistent. Instead, they have what is referred to as *weak memory models* which do not require that instructions take effect in program order. This allows the hardware to optimize performance by limiting the number of accesses to the shared memory.

To date, there is limited research on verifying linearizability on hardware weak memory models. Existing approaches [7], [15], [18], [28] adopt an *architectural model* of either the implementation or both the specification and implementation, i.e., a model including features of the underlying architecture. This is not ideal for two reasons. Firstly, the details of many commercial architectures are not available publicly. Secondly, these approaches require the method for proving linearizability to be modified, and hence existing proof methods and their tool support cannot be used.

In recent work [14], we presented a definition of linearizability that can be used across a variety of weak memory models. The definition is essentially the same as that of standard linearizability, but is based on the *observable* order of program statements rather than the order they appear in the program code. This order can be derived via systematic testing (e.g., see [21]) without an understanding of the underlying architecture.

To use this definition of linearizability, we require *observational models* of program behaviour. Such models abstract from the underlying architecture, and instead model a program as if it were running on a sequentially consistent architecture but with the behaviour allowed by the weak memory model. Hence, existing proof methods for sequentially consistent architectures can be used. In this paper, we provide examples of such observational models; specifically, for the weak memory models TSO [24], [27] and XC [27]. We describe the use of these models in checking linearizability with the NuSMV model checker following the existing proof method of Derrick et al. [11]–[13], [23] and its associated model checking support [25]. In Section II we introduce the concept of linearizability and our running example, the Linux reader-writer mechanism seqlock. In Section III we discuss weak memory models using TSO as an example, and in Section IV we describe the general form of an observational model for weak memory models. We provide specific models for TSO in Section V and for the weaker memory model XC in Section VI. We apply the approach to seqlock and summarise our model checking results in Section VII. We review related work in Section VIII before concluding in Section IX.

II. Linearizability

Concurrent objects are objects that are developed to be used in a multi-threaded environment. 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) 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.

```
x1 = 0, x2 = 0;
write(in: d1,d2)        read(out: d1,d2)
{  x1 = d1;             {    d1 = x1;
   x2 = d2; }                d2 = x2; }
```

Fig. 1. seqlock specification

```
 x1 = 0, x2 = 0;          read(out: d1,d2) {
 c = 0, lock = 1;            word c0;
                             do{
 write(in: d1,d2) {             do{
1   acquire;              7         c0 = c;
2   c++;                  8       }while (c0%2!=0);
3   x1 = d1;              9       d1 = x1;
4   x2 = d2;              10      d2 = x2;
5   c++;                  11     }while (c != c0);
6   release;              12     return(d1,d2);
}                         }
```

Fig. 2. seqlock implementation [7]

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 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 [19] is the standard correctness criterion for verifying concurrent objects such as seqlock. It is used to relate each *history*, i.e., allowed 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 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 [19] and a number of approaches have been developed for proving it along with associated tool support [6], [8], [11]–[13], [17], [23], [30]. In particular, Derrick et al. [11]–[13], [23] 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

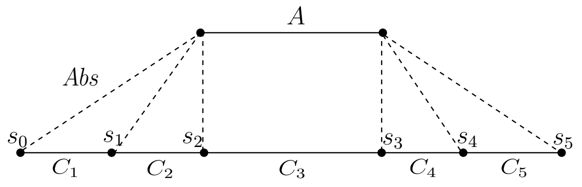

Fig. 3. Simulation-based proof method for Linearizability

code) must be related to either the pre- or post-state of A via an abstraction relation Abs. In Fig. 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. These assertions ensure the thread simulates A as required. To ensure threads do not interfere with each other's behaviour, an additional proof step checks that each step does not change shared variables in such a way that any of the required assertions for another thread can be broken.[1] 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 [12], [23] and, being thread-local and step-local, lends itself to automation using a model checker [25]. Unlike other model checking approaches for linearizability, the results are not restricted to a fixed number of threads, or particular sequences of operation calls.

III. 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 [24], [27].

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 from the memory if the buffer is empty. This is referred to as *bypassing*. 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 as if it occurred before the store.

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

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

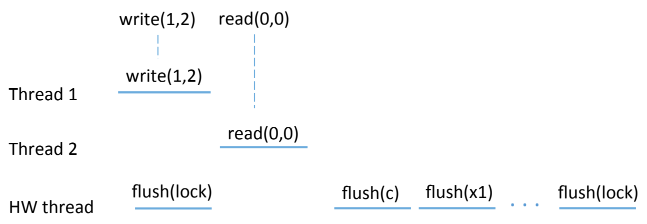

Fig. 4. Linearizability fails on TSO

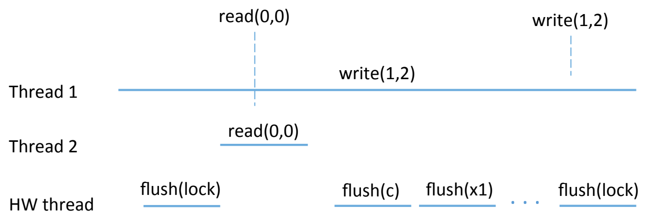

Fig. 5. Linearizability succeeds on TSO

A typical situation is illustrated for seqlock in Figure 4 which 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. 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 linearizability cannot be proved.

However, it could be 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. 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.

Derrick et al.'s proof method can be used with this concept by considering the return of an operation to be the point in the execution where all threads can observe that the operation has completed (e.g., the final flush of an operation that writes to global variables in the case of TSO).

IV. Observational models

We model each operation of a concurrent object by a transition system where each program statement (labelled by a line number) is modelled as a state transition in a standard fashion, subject to the following provisions: Firstly, statements that modify a global variable are split into two steps, one for loading the variable into a register (local to the thread), and one for modifying the register and storing the result back into the global variable (e.g., c++ is modelled as $local_c = c$; $c = local_c + 1$, where $local_c$ is a register). This ensures we allow all possible interference between threads. Secondly, 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.

Branch statements are often not treated as instructions but simply as control flow directives [24], [27]. However, to eventually model more complex weak memory models, like ARM and Power [4] (where speculative execution is supported), it seems beneficial to treat branching as a statement (and hence as a separate transition) [10]. To provide a uniform treatment we follow this idea, and model a branch by two transitions, one for the true case and one for the false case. Loops can also be modelled by two transitions, one for entering the loop and one for exiting the loop. We also treat *invocations* and *return* statements as separate transitions.

Hence, we consider *loads*, *stores*, *branch* and *return* statements, and *invocations*, as well as some architecture specific statements like *fences* and atomic *read-modify-writes (RMW)* (e.g., the *compare-and-swap* (CAS) instruction [27]).

The deviation in the behaviour of a program under a weak memory model is often described in terms of the *observations* that can be made of values read from and written to global variables (the former by examining local registers [21], the latter by examining the shared memory). Under a sequentially consistent architecture the observations follow the *program order* that is prescribed by the code, i.e., a (control flow) graph $<_P = (\mathcal{L}_P, next_P)$ over the set of instruction labels $\mathcal{L}_P$ and a left-total next-step relation $next_P : \mathcal{L}_P \times Bool \to \mathcal{L}_P$, which maps each label to one or more successors. For more than one successor, the second argument specifies a condition to determine the flow of control; when a label has only one successor it equals true and is often omitted. The root node of $<_P$ is 0 which models the idle state, and is the successor for the labels of return statements.

Under weak memory models the order of observations is weaker than $<_P$ and often described as (possible) *reordering* of statements (e.g., [27]). A weaker order presents itself through the observations of some statements being un-ordered, and hence occurring in any order. This weaker order can be described as a set of graphs where each is a restriction of $<_P$ to a subset of labels which are ordered (omitting the un-ordered ones). We define $<_P|_S = (S, next_S)$ as the graph $<_P$ restricted to labels in S such that for all $l \in S, b \in \mathbb{B}$, $next_S(l, b) = l'$ with l' being the first state in S reachable from l in $<_P$. We assume that the set S contains all branch points in order to maintain the branching structure.

Furthermore, 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 global variable. This enables us to model *bypassing* as well as the delayed observation of store steps.

We denote the sets of labels (i.e., line numbers) as follows: $\mathcal{L}_{ld}$ (*loads*), $\mathcal{L}_{st}$ (*stores*), $\mathcal{L}_{br}$ (*branches*), $\mathcal{L}_{ret}$ (*returns*), $\mathcal{L}_f$ (*fences*), $\mathcal{L}_{rmw}$ (*RMWs*), and $\mathcal{L}_{inv}$ (the invocation label, 0).

In the following sections, we consider observational models for the well-known TSO memory model and the theoretical XC memory model for which we assume given reordering constraints from the literature [27]. For each memory model M, we provide its observation order $<_{P(M)}$ for a program P. $<_{P(M)}$ prescribes the control flow for P under memory model M thereby modelling its possible behaviour. We then provide

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 I
ORDER CONSTRAINTS FOR TSO ARCHITECTURES

a set of rules for translating a given program into a transition system that is consistent with $<_{P(M)}$.

V. A MODEL OF TSO

In [27], Sorin et al. describe the effects of TSO as a reordering of statements as summarised in Table I. 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 III). From the information in this table, we can deduce the observation order $<_{P(TSO)}$ and based on that provide an explicit model of program behaviour under TSO. Such a model can be verified using linearizability [14].

From the table we can see that stores may be reordered with subsequent loads, but their results will be locally visible in program order. Therefore, *l-stores* follow the program order while *g-stores* follow a separate order but have to come after their corresponding *l-stores*. 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. For branch statements (which are not included in the table) we assume that they are not reordered with respect to loads, fences and RMWs but stores can be reordered after subsequent branches. This assumption is consistent with other models of TSO, e.g., [2], [29].

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

- a *local order* which orders steps with a local effect; it is identical to $<_P$, i.e., $<_L = <_P$
- a *store order* which orders steps with a global effect like *g-stores*, fences and RMWs; it also includes branches and invocations since they cannot be reordered *after* stores: $<_S = (\mathcal{L}_S, next_S)$ with $\mathcal{L}_S = \mathcal{L}_{st} \cup \mathcal{L}_{br} \cup \mathcal{L}_f \cup \mathcal{L}_{rmw} \cup \mathcal{L}_{inv}$ and $next_S : \mathcal{L}_S \times (\mathcal{F}_B \cup \{true\}) \to \mathcal{L}_S$ defined similarly to $next_P$ but where, instead of branch conditions b as in $next_P$, $\mathcal{F}_B$ contains flags r_b which indicate the branch evaluation that must have occurred earlier.

We also have a *bypass order*, $<_B$, which links $<_L$ and $<_S$ by enforcing that each *l-store* occurs before the corresponding *g-store*. This order synchronises between $<_L$ and $<_S$.

The overall observable order, $<_{P(TSO)}$, describes the control flow as the union of *load-*, *store-* and *bypass-order*, i.e., $<_{P(TSO)} \;=\; <_L \cup <_S \cup <_B$.

A. Transition System Model

The order of transitions in a transition system is typically enforced by a counter whose values relate unambigously 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 two of the orders that need to be maintained: 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 $<_S$. $<_B$ is enforced through a combination of values for the pc_S and the status of a corresponding local register variable (see below). Each transition is guarded by the counter(s) that enforces the order(s) which they are part of.

Assume the ordering graphs $<_L$ and $<_S$, global variable v_g with its local register r_g, and local variable v_l. We model r_g as a sequence of entries to keep track of all *l-stores* to v_g, including their order of occurrence, until the corresponding *g-stores* store these entries in global memory (following the order of occurrence of the *l-stores*). The transitions for each statement type take the following form.

Invocation. 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 the *g-stores* which have not occurred are from a previous occurrence of the same operation then we need to also keep the previous operation's input values. Hence, the inputs of an operation are also modelled as sequences of values, each representing the inputs of one occurrence of the operation.

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*, when the operation has inputs.

$$\begin{aligned} Invoke(v) == \;\; & pc_L = 0 \wedge pc_L' = next_L(0) \wedge \\ & pc_S' = pc_S \frown \langle next_S(0) \rangle \wedge in' = in \frown \langle v \rangle \end{aligned}$$

Load. A load can occur when pc_L has reached its label, and upon occurrence it sets pc_L to the next label in the load order and reads the value of a variable. 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 shared memory, namely v_g. Assume $last(s)$ denotes the last element of a sequence s. Then given a line `n:` $\texttt{v}_l\texttt{=e(v}_g\texttt{);}$ (where $\texttt{e(v}_g\texttt{)}$ is an expression in terms of $\texttt{v}_g$) modelling the merge of a load and a modification to the loaded value, we have the following transition.

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

Local store. An *l-store* can occur when pc_L has reached its label. Upon occurrence it sets pc_L to the next label in the load order and appends the new value to the corresponding register variable r_g. Given the line `n: v_g = val;` we have

$$\textit{l-store}\,(n) == pc_L = n \wedge pc'_L = next_L(pc_L) \wedge \\ r'_g = r_g \frown \langle val \rangle$$

Global store. 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 (to enforce $<_B$ for each (*l-store*, *g-store*) pair). 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 $next_S(n, c)$. When $next_S(n, c) = 0$ and all labels included in $<_S$ have been executed (i.e., the stores of the current operation are finished), the step removes the head of pc_S (so that the first *g-store* of the next operation can occur if there is any). Given the line `n: v_g = val;` we have

$$\textit{g-store}\,(n) == head(pc_S) = n \wedge r_g \neq \langle\,\rangle \wedge \\ v'_g = head(r_g) \wedge r'_g = tail(r_g) \wedge \\ pc'_S = (\textbf{if } next_S(n,c) \neq 0 \textbf{ then } \langle next_S(n,c)\rangle \frown tail(pc_S) \\ \textbf{else } tail(pc_S))$$

Branch. A *branch* statement is captured by two transitions, one of type *Branch* and one of type *SBranch*. A transition of the first type is executed when the pc_L has reached its label. It evaluates its condition b and sets a corresponding Boolean flag r_b. It updates pc_L to its next statement to be executed. This flag r_b will direct the flow in $<_S$ along the same branch as in $<_L$. Note that for this transition the next label depends on the evaluated condition.

$$Branch(n) == pc_L = n \wedge pc'_L = next_L(n, b) \wedge r'_b = b$$

An *SBranch* is the corresponding flow directive in the store order graph. It must occur after the branch statement itself (in the load order graph) has occurred. It is enabled when the head of pc_S has reached the label of the branch and pc_L has a value greater than the label (and hence the branch condition has been evaluated and the branch flag set). Assume b is the branch at label n and r_b the branch flag:

$$SBranch(n) == head(pc_S) = n \wedge \ n <_L pc_L \ \wedge \\ pc'_S = (\textbf{if } next_S(n, r_b) \neq 0 \textbf{ then } \langle next_S(n, r_b)\rangle \frown tail(pc_S) \\ \textbf{else } tail(pc_S))$$

Fence and RMW. A fence at line n means $n \in <_L \cap <_S$ i.e., the label occurs in both orderings. To enable this step requires that pc_L as well as pc_S are set to n. The latter enforces that all *g-stores* before the fence have occurred (and hence all registers are empty). The transition updates pc_L as well as pc_S to their next values. An RMW step is similar, but has a guard and, depending on the guard's value, may modify a global variable.

$$Fence(n) == pc_L = n \wedge head(pc_S) = n \wedge \\ pc'_L = next_L(pc_L) \wedge \\ pc'_S = (\textbf{if } next_S(n) \neq 0 \textbf{ then } \langle next_S(n)\rangle \frown tail(pc_S) \\ \textbf{else } tail(pc_S))$$

Return. The return step of an operation (if any) can occur when the pc_L has reached its label $n \in \mathcal{L}_{ret}$.

$$Return(n) == pc_L = n \wedge pc'_L = 0$$

B. The seqlock example

As an example, we show how seqlock would be modelled observationally. Consider the `write` operation. From the program code we can derive the following load and store flow graphs (here denoted as a sequence as there is no branching and all conditions in $next_L$ and $next_S$ are *true*), $<_L = \langle 0, 1, 2, 22, 3, 4, 5, 52, 6\rangle$[2] and $<_S = \langle 1, 22, 3, 4, 52, 6\rangle$. We define a transition for each line of code within `write` denoting these $W0$, $W1$, etc. as shown in Table II.

$$\begin{aligned}
&(*\ Invoke_{write}\ *)\\
&W0 == pc_L = 0 \wedge pc'_L = 1 \wedge pc'_S = pc_S \frown \langle 1\rangle \wedge\\
&\qquad in'_1 = in_1 \frown \langle d1\rangle \wedge in'_2 = in_2 \frown \langle d2\rangle\\
&(*\ RMW\ *)\\
&W1 == pc_L = 1 \wedge lock = 1 \wedge head(pc_S) = 1 \wedge\\
&\qquad pc'_L = 2 \wedge lock' = 0 \wedge pc'_S = \langle 22\rangle \frown tail(pc_S)\\
&(*\ Load\ and\ l\text{-}store\ \ *)\\
&W2 == pc_L = 2 \wedge pc'_L = 22 \wedge\\
&\qquad local'_c = (\textbf{if } r_c = \langle\,\rangle \textbf{ then } c + 1 \textbf{ else } last(r_c) + 1)\\
&W22 == pc_L = 22 \wedge pc'_L = 3 \wedge r'_c = r_c \frown \langle local_c\rangle\\
&W3 == pc_L = 3 \wedge pc'_L = 4 \wedge r'_{x1} = r_{x1} \frown \langle last(in_1)\rangle\\
&\quad \ldots\\
&W6 == \ pc_L = 6 \wedge pc'_L = 0 \wedge r'_{lock} = r_{lock} \frown \langle 1\rangle\\
&(*\ g\text{-}stores\ \ *)\\
&W2s == head(pc_S) = 22 \wedge r_c \neq \langle\,\rangle \wedge\\
&\qquad c' = head(r_c) \wedge r'_c = tail(r_c) \wedge pc'_S = \langle 3\rangle \frown tail(pc_S)\\
&\quad \ldots
\end{aligned}$$

TABLE II
TRANSITIONS MODELLING THE `write` OPERATION

$W0$ 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. $W1$ requires that both pc_L and pc_S are 1. It updates both to the next position in their order and locks the lock. Note that the `acquire` command is a RWM with a built-in fence. $W2$ 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 $local_c$. $W22$ adds the value of $local_c$ to the register (similarly, $W5$ and $W52$). $W3$ (and similarly $W4$) do not need bypassing since they are storing values from local (input) variables. $W6$ appends the value 1 to the lock register and modifies pc_L to 0 as it is the last step in the `write` operation. The remaining transitions for `write` are the *g-stores* of the (potentially delayed) stores. Before performing a *g-store* a check is required as to whether the *l-store* to the register has occurred (and the register is therefore not empty).

Since the `read` operation does not write to any global variable there are no further store steps that may be delayed.

[2]The steps in line 2 and 5 are not atomic and performed in two steps, a load and a store step. We introduce auxiliary line numbers 22 and 52, respectively.

The `read` operation just performs loads with bypassing which are modelled similarly to the loads of the `write` operation.

VI. A model for XC

XC [27] is a (theoretical) weak memory model that is weaker than TSO, allowing reorderings similar to more complex memory models such as Power and ARM [4]. In contrast to ARM and Power, however, XC does not support *speculative execution* (i.e., the reordering of instructions within a branch with its branch statement [27]), nor *non-multi-copy atomicity* where written global variables are not visible to all processors at the same time. An XC model is hence a stepping stone towards a more complex model for programs running on ARM or Power architectures.[3]

The observable effects of potential reordering in XC are summarised in Table III (taken from [27]). Again, X denotes an enforced ordering and B that bypassing is required if the commands are to the same address, i.e., the same global variable. The entry A denotes that ordering is enforced only if the commands are to the same address.

As in TSO, registers for each global variable are modelled as sequences in XC. We assume branches behave like a branch together with a control fence (*cfence*) in the ARM/Power architecture [5] (an `isync` on Power, and an `isb` on ARM), i.e., branches cannot be reordered with loads and stores later in the program order, nor with loads and stores, on which the branch depends, earlier in the program order (but branches can be reordered with each other) [10]. Since in XC stores to one variable can be reordered before loads of another variable, we need to also consider *data dependencies* between loads and stores: when a store depends on a variable that has been loaded before (in program order) then this store cannot occur before the load of the variable it depends on [5].

To determine the memory order we follow similar reasoning to that in Section V: both loads and stores follow the program order only if they refer to the same address, resulting in a load and store order per global variable. We denote with $\mathcal{L}_t(a)$ labels of type t in the program-order that refer to address a. Note that fences, invokes and returns are not address specific, whereas branches might depend on more than one address, (i.e., $l \in \mathcal{L}_{br}(a)$ for more than one a). For each address a we have

- an *a-load* order which orders all statements other than *g-stores* referring to address a: $<_L^a = <_P |_{\mathcal{L}_L(a)}$. where $\mathcal{L}_L(a) = \mathcal{L}_{ld}(a) \cup \mathcal{L}_{ls}(a) \cup \mathcal{L}_{rmw}(a) \cup \mathcal{L}_{br}(a) \cup \mathcal{L}_f \cup \mathcal{L}_{ret} \cup \mathcal{L}_{inv}$.
- an *a-store* order, $<_S^a$, which orders the *g-stores* and RMWs to address a as well as all fences. Since *g-stores* are not ordered, unless they are to the same address, this is easily captured by modelling the registers for each address as sequences that behave like FIFO buffers. In particular, branches are not needed as part of the order (in TSO they were required to distinguish which branch the program had taken, and hence the order in which stores *on different addresses* had occurred).
- a *bypass* order, $<_B$, which orders the *l-store* and the *g-store* of each store.

The overall observable order, $<_{P(XC)}$, describes the control flow as the union of *load-*, *store-* and *bypass-order* for all addresses a, i.e., $<_{P(XC)} = \bigcup_{a \in Addr} (<_L^a \cup <_S^a) \cup <_B$.

[3]The most recent release of ARM, v8.0, is now multi-copy atomic [22], and thus is closer to XC.

XC		Command 2			
		load	store	RMW	fence
Command 1	load	A	A	A	X
	store	B	A	A	X
	RMW	A	A	A	X
	fence	X	X	X	X

TABLE III
Order constraints for XC architectures

A. Transition System Model.

To capture these constraints in a transition system, the load order is enforced by a separate counter for each address, i.e., $pc_L : Addr \to \bigcup_{a \in Addr} \mathcal{L}_L(a)$. The local state space consists of register variables (for each global variable) which are sequences (modelling a FIFO buffer per address a). Since the stores to different addresses are not ordered we do not require a variable pc_S as for TSO, nor do we need the flags r_b (since branches are not part of the store orders, as mentioned above).

In the following we denote with v_a the global variable at address a, with r_a the register for storing values to v_a, and with v_l a local variable. Assume an a-load order of $<_L^a = \langle \mathcal{L}_L(a), next_L^a \rangle$ for every global address a.

Invocation. An invocation transition updates pc_L for each a to the first element of the load order $<_L^a$.

$$\begin{aligned} Invoke(v) == \forall\, a : Addr \cdot (&pc_L(a) = 0 \wedge \\ &pc_L(a)' = next_L^a(pc_L(a))) \wedge \\ &in' = in \frown \langle v \rangle \end{aligned}$$

Load. A load from address a at line number n is of the same form as *Load* in the TSO model, it updates pc_L at index a to the next value as given in $<_L^a$.

$$\begin{aligned} Load_a(n) == {} & pc_L(a) = n \wedge pc_L(a)' = next_L^a(pc_L(a)) \wedge \\ & v_l' = (\textbf{if } r_a = \langle\,\rangle \textbf{ then } e(v_a) \textbf{ else } e(last(r_a))) \end{aligned}$$

Local store. An *l-store* to address a at line n appends a value *val* to the register variable r_a if $pc_L(a)$ equals n.

$$\begin{aligned} l\text{-}store_a(n) == {} & pc_L(a) = n \wedge pc_L(a)' = next_L^a(pc_L(a)) \\ & \wedge r_a' = r_a \frown \langle val \rangle \end{aligned}$$

Data-dependent local store. If an *l-store* at line n stores a register value r_{val} which (in the program order) was previously set by a load from a different address b (which is ordered by $<_L^b$) at line m (i.e, $m <_P n$), an additional guard is introduced to ensure the *data-dependency* is preserved: $pc_L(b)$ is required to have proceeded past the load from address b.

$$\begin{aligned} & data\text{-}dependent\text{-}l\text{-}store_a(n, b, m) == \\ & \quad pc_L(a) = n \wedge (m <_L^b pc_L(b)) \wedge \\ & \quad pc_L(a)' = next_L^a(pc_L(a)) \wedge r_a' = r_a \frown \langle r_{val} \rangle \end{aligned}$$

Global store. A *g-store* to v_a can occur whenever register r_a is not empty (enforcing $<_B$), and on occurring writes the head of the register to v_a and removes it from the sequence.

$$g\text{-}store_a == \quad r_a \neq \langle\,\rangle \wedge v'_a = head(r_a) \wedge r'_a = tail(r_a)$$

Branch. A *branch* statement is executed when $pc_L(a)$, for all a that the branch depends on, has reached its label. It evaluates its condition b and updates all pc_L to their next statement to be executed.

$$Branch_{as}(n) == \forall\, a : as \cdot pc_L(a) = n \wedge pc'_L(a) = next^a_L(n, b)$$

RMW. An RMW step to address a at line n requires that the pc_L at index a is set to n and that the register for a, r_a, has been flushed. The step updates the $pc_L(a)$ to the next value in $<^a_L$, and modifies the global variable v_a.

$$\begin{aligned} RMW_a(n) == {} & pc_L(a) = n \wedge r_a = \langle\,\rangle \wedge \\ & pc_L(a)' = next^a_L(pc_L(a)) \wedge v'_a = val \end{aligned}$$

Fence. A fence command at line n is enabled when the pc_L has reached n for all addresses and all registers are flushed. It increases the pc_L for each address according to the corresponding load order.

$$\begin{aligned} Fence(n) == \forall\, a : Addr \cdot (& pc_L(a) = n \wedge r_a = \langle\,\rangle \wedge \\ & pc_L(a)' = next^a_L(pc_L(a))) \end{aligned}$$

Return. The return step needs to capture that for all a, $pc_L(a)$ has reached its label.

$$Return(n) == \forall\, a : Addr \cdot (pc_L(a) = n \wedge pc_L(a)' = 0)$$

This modelling scheme can be applied to a specific example in a similar fashion to that for TSO. Note that the resulting transition system is significantly simpler than the corresponding TSO model. It has less ordering constraints and instead more non-determinism.

VII. Model checking on weak memory models

The observational models of Sections V and VI capture the behaviour allowed by the respective weak memory models when run in a sequentially consistent manner. Hence, to check whether a modelled concurrent object is linearizable, we can use exisiting proof methods and tools for checking linearizability. In particular, the state transition models can be used directly with the thread-local, step-local proof method of Derrick et al. [11]–[13], [16], [23].

Smith [25] defines a model checking approach for this proof method which is encoded in the TLA+ model checker, TLC. In this approach, the model is restricted to perform only one step at a time, which is nondeterministically chosen (and hence all steps are checked via exhaustive search). The pre-states of each chosen step are constrained by assertions (based on Derrick et al.'s approach) and the reachable post-states are then checked to ensure that the assertions are maintained when the step is performed. Two checks are done in this way: one to show that each step simulates an abstract operation (if it is a linearization step) or an abstract skip (otherwise); and one to show that the step does not invalidate the assertions of another arbitrary thread (and hence there is no interference between threads). A check is also performed to show that the required assertions hold in the initial state of the modelled system.

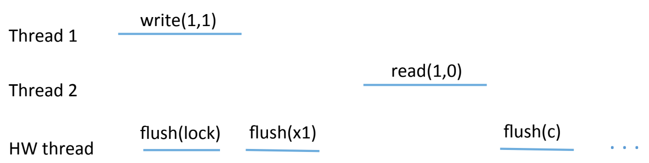

Fig. 6. Linearizability fails on XC

For our experiments, we adopted the approach of Smith [25] and chose linearization points based on the approach of [16]. Instead of TLC, we used the symbolic model checker NuSMV [9] as the tool handles the large assertions required for models on weak memory very 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 all three checks the model checking returns within a few seconds on seqlock modelled in TSO and in XC. 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. 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 is operates correctly.[4] On XC, 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 6. The write by *Thread*1 to `x1` has been flushed before its write to `c` allowing *Thread*2 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 line 3 and after line 4. 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`) and before line 2 (to ensure that the first increment of `c` cannot happen before `acquire`). 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.

VIII. Related work

Modelling the behaviour of programs under hardware weak memory models has been investigated by others (e.g., [1]–[3], [20], [29]). While some of this work presents very similar transition systems (in particular the two control flow graphs and the split of stores into two steps is similar in [2], [29]), none of the approaches uses the notion of linearizability to verify correctness. Linearizability, however, has the advantage over other approaches that it has a proof method that is not only thread-local but also step-local. Although the user of such

[4]Note that correctness in this case is in the context of an *operation-race free* client; see [16] for details.

a method has to provide assertions (see Section II) 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. The assertions can be generated for programs running on sequentially consistent architectures [26] and it is future work to extend this approach to programs under weak memory models. In [20] the similarly 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.

IX. Conclusions

This paper has presented observational models for the TSO and XC weak memory models. The main advantage of these models over previously published architectural models is that they can be used directly with existing proof methods for linearizability. They can also be used with commercial weak memory models whose architecture cannot be modelled since the details are not available publicly. In such cases, observations of values in shared memory and local registers can be made during systematic testing. It is these observations that form the basis of our observational models.

Our work so far is limited to weak memory models whose behaviour can be represented in terms of instruction reordering. More complex weak memory models such as those of the IBM Power and ARM architectures allow additional behaviour due to (1) supporting *speculative execution*, where instructions after a branch instruction may be executed before the branch condition has been determined, and (2) being *non-multi-copy-atomic*, meaning a store by one thread may propagate to other threads at different times. Incorporating them into our models is an ongoing area of work.

Acknowledgement This work was supported by Australian Research Council Discovery Grant DP160102457.

References

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

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

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

[4] 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 *DAMP '09*, pages 13–24. ACM, 2008.

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

[6] D. Amit, N. Rinetzky, T.W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV 2007*, volume 4590 of *LNCS*, pages 477–490. Springer, 2007.

[7] S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In *ESOP 2012*, volume 7211 of *LNCS*, pages 87–107. Springer, 2012.

[8] C. Calcagno, M. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In *SAS 2007*, volume 4634 of *LNCS*, pages 233–238. Springer, 2007.

[9] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *CAV 2002*, volume 2404 of *LNCS*. Springer, 2002.

[10] R.J. Colvin and G. Smith. A wide-spectrum language for verification of programs on weak memory models. In *FM 2018*, LNCS. Springer, 2018.

[11] J. Derrick, G. Schellhorn, and H. Wehrheim. Proving linearizability via non-atomic refinement. In *IFM 2007*, volume 4591 of *LNCS*, pages 195–214. Springer, 2007.

[12] 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.

[13] J. Derrick, G. Schellhorn, and H. Wehrheim. Verifying linearisabilty with potential linearisation points. In *FM 2011*, volume 6664 of *LNCS*, pages 323–337. Springer, 2011.

[14] J. Derrick and G. Smith. An observational approach to defining linearizability on weak memory models. In *FORTE 2017*, LNCS. Springer, 2017.

[15] J. Derrick, G. Smith, and B. Dongol. Verifying linearizability on TSO architectures. In *iFM 2014*, volume 8739 of *LNCS*, pages 341–356. Springer, 2014.

[16] S. Doherty and J. Derrick. Linearizability and causality. In *Software Engineering and Formal Methods - 14th International Conference, (SEFM 2016)*, volume 9763 of *LNCS*, pages 45–60. Springer, 2016.

[17] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE 2004*, volume 3235 of *LNCS*, pages 97–114. Springer, 2004.

[18] 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.

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

[20] O. Lahav and V. Vafeiadis. Owicki-Gries reasoning for weak memory models. In *Automata, Languages, and Programming*, pages 311–323. Springer, 2015.

[21] 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.

[22] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell. Simplifying ARM concurrency : multicopy-atomic axiomatic and operational models for ARMv8. In *POPL 2018*. ACM Press, 2018.

[23] 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.

[24] 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.

[25] G. Smith. Model checking simulation rules for linearizability. In *SEFM 2016*, volume 9763 of *LNCS*, pages 188–203. Springer, 2016.

[26] G. Smith and J. Derrick. Invariant generation for linearizability proofs. In *SAC 2016 ACM*, pages 1694–1699. ACM, 2016.

[27] 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.

[28] O. Travkin, A. Mütze, and H. Wehrheim. SPIN as a linearizability checker under weak memory models. In *HVC 2013*, volume 8244 of *LNCS*, pages 311–326. Springer, 2013.

[29] O. Travkin and H. Wehrheim. Verification of concurrent programs on weak memory models. In *ICTAC 2016*, pages 3–24. Springer, 2016.

[30] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.