Concurrent Library Correctness on the TSO Memory Model

Sebastian Burckhardt[1], Alexey Gotsman[2], Madanlal Musuvathi[1], and Hongseok Yang[3]

[1] Microsoft Research
[2] IMDEA Software Institute
[3] University of Oxford

Abstract. Linearizability is a commonly accepted notion of correctness for libraries of concurrent algorithms. Unfortunately, it is only appropriate for sequentially consistent memory models, while the hardware and software platforms that algorithms run on provide weaker consistency guarantees. In this paper, we present the first definition of linearizability on a weak memory model, Total Store Order (TSO), implemented by x86 processors. We establish that our definition is a correct one in the following sense: while proving a property of a client of a concurrent library, we can soundly replace the library by its abstract implementation related to the original one by our generalisation of linearizability. This allows abstracting from the details of the library implementation while reasoning about the client. We have developed a tool for systematically testing concurrent libraries against our definition and applied it to several challenging algorithms.

1 Introduction

Concurrent software developers nowadays rely heavily on libraries of concurrency patterns and high-performance concurrent data structures, such as `java.util.concurrent` for Java and Intel's Threading Building Blocks for C++. The algorithms implemented by these libraries are very efficient, with the downside being that they are notoriously difficult to design and implement. More surprisingly, it is often difficult to understand even what it means for them to be correct! Correctness of concurrent libraries is commonly formalised by the notion of *linearizability* [11], which fixes a certain correspondence between the library and its abstract specification, the latter usually sequential, with methods implemented atomically. Unfortunately, the classical definition of linearizability is only appropriate for sequentially consistent (SC) memory models, in which accesses to shared memory occur in a global-time linear order. At the same time, most multiprocessors (x86 [15], Power [17], ARM [1]) and programming languages (Java [12], C++ [2]) provide weaker memory models that allow more efficient implementations at the expense of exhibiting counterintuitive behaviours in some cases.

In this paper, we present the first definition of linearizability on a weak memory model, Total Store Order (TSO), implemented by x86 processors [15] (Section 4). We show that our definition is a correct one in the sense that it validates what we call the Abstraction Theorem: while proving a property of a client of a concurrent library, we can soundly replace the library by its abstract implementation related to the original one by

H. Seidl (Ed.): ESOP 2012, LNCS 7211, pp. 87–107, 2012.

our generalisation of linearizability (Theorem 4, Section 5). The abstract implementation is usually simpler than the original one, with commands executing at a coarser grain of atomicity. The Abstraction Theorem thus formalises the intuitive requirement for a good definition of linearizability, which is that the library should provide an illusion of such a simpler atomic implementation. It also has a practical value as a compositional verification technique: it allows abstracting from the details of the library implementation while reasoning about its client, despite subtle interactions between the two caused by the weak memory model. As a corollary of the Abstraction Theorem, we establish that the proposed notion of linearizability is compositional (Corollary 5, Section 5).

To demonstrate that our notion of linearizability is appropriate for practical concurrent algorithms, we have developed a tool for systematically testing such algorithms against the definition and applied it to several examples (Section 6). We have also proved the linearizability of one of the algorithms formally (Theorem 3, Section 4). The algorithms considered are challenging to reason about and to specify, as they sometimes exhibit behaviours not reproducible on a sequentially consistent memory model.

The TSO Memory Model. The most intuitive way to explain the TSO memory model is operationally (Section 2), using an abstract multiprocessor machine in which every CPU has a *store buffer*. The buffer holds write requests that were issued by the CPU, but have not yet been *flushed* into the shared memory. A command that would like to write to a location in memory stores the corresponding write request in the store buffer of the CPU executing it, thus avoiding the need to block the CPU while the write completes.The CPU may decide to flush a store buffer entry into the main memory at any time, subject to maintaining the FIFO ordering of the buffer: the oldest write will be flushed first. A command that would like to read from a location in memory returns the value stored in the newest entry for this location in the store buffer of the CPU executing it; if such an entry does not exist, it accesses the memory directly.

The behaviour of programs running on TSO can sometimes be counterintuitive. For example, consider two memory locations `x` and `y` initially holding 0. On standard x86 processors, if two CPUs respectively write 1 to `x` and `y` and then read from `y` and `x`, as in the following program, it is possible for both to read 0 in the same execution:

$$\begin{array}{c} \texttt{x = y = 0;} \\ \texttt{x = 1; b = y;} \quad \| \quad \texttt{y = 1; a = x;} \\ \{\texttt{a} = \texttt{b} = 0\} \end{array}$$

This outcome cannot happen on a sequentially consistent machine, where both reads and writes access the memory directly. On TSO, it happens when the reads from `y` and `x` occur before the writes to them have propagated from the store buffers of the corresponding CPUs to the main memory. To exclude such behaviours, TSO processors provide special instructions, called *memory barriers*, that force the store buffer of the corresponding CPU to be flushed completely before executing the next instruction. Adding memory barriers after the writes to `x` and `y` in the above program would make it produce only SC behaviours. However, barriers incur a performance penalty.

Technical Challenges. The presence of store buffers leads to subtle interactions between a library and its client that make it challenging to define linearizability. Showing

linearizability requires us to provide, for every execution of the concrete library implementation, an execution of the abstract library interacting with the client in a similar way (in a certain technical sense). Interactions between the library and the client are usually defined in terms of *histories*, which, in the classical definition, are sequences of calls to and returns from the library, along with the values passed. In the case of TSO, however, this would not describe all interactions between the two components, since one of them can exhibit a side effect on the other via a store buffer. For example, a memory barrier inside a library method will flush entries written there by client as well as library code. More subtly, write commands in a library method can insert entries into the store buffer without ensuring that they get flushed by the time the method returns. For this reason, on TSO, the method return point does not characterise the time by which the effects of these writes will be visible to the client (see the seqlock example in Section 4). To define the notion of linearizability on TSO that validates the Abstraction Theorem and is compositional, we thus need histories to describe the information relevant to the client about how the library uses store buffers. The classical notion of linearizability [11], which is not aware of store buffers, cannot specify this.

Main Ideas. Our main insight lies in identifying the additional information that we need to record in histories to get a definition of linearizability on TSO validating the Abstraction Theorem. Namely, the contents of a store buffer can be viewed as a sandwich consisting of blocks of entries inserted there by an invocation of a library method or a fragment of the client code between two such invocations. We show that the behaviour of the library with regards to the store buffer that can affect the client is completely described by the moments of time at which the first and the last elements of any given library layer in the sandwich get flushed. Roughly speaking, the time when a library layer starts to get flushed defines an assumption the library makes about the client: since store buffers are FIFO, the library requires the previous client layer in the buffer to be flushed completely before this. The time by which a library layer is flushed completely represents a guarantee the library provides to the client: this action enables the next client layer to be flushed starting from this point of time.

To specify this, we enrich histories with additional actions denoting the times when a layer of entries inserted by every library method invocation starts to get flushed and is flushed completely. Linearizability then requires preserving the order between some of these actions in a history of the concrete library implementation when providing a matching history of the abstract library implementation. As we show, this is sufficient to establish the Abstraction Theorem.

The proposed definition of linearizability on TSO requires a novel way of specifying libraries. In the classical definition, the specification of a library method often consists of one atomic action. Since on TSO writes can be delayed in the store buffer, such a specification according to our notion of linearizability is often given by two atomic actions: one that atomically writes entries into the store buffer, and one that flushes them into the memory, possibly after the method returns. The resulting specification captures the effects of using the store buffer visible to the client, yet is simpler than the implementation: it ensures that all the locations written to by a library method will be written to the memory atomically, albeit at some later time. We provide examples of such specifications in Section 4 and [5, Appendix B].

2 TSO semantics

In this section, we present the operational semantics of the TSO memory model, following [15], along with our modifications to it needed to define linearizability.

Notation. We write A^+ and A^* for the sets of all nonempty, respectively, possibly empty finite sequences of elements of a set A. We denote the empty sequence with ε and the concatenation of sequences α_1 and α_2 with $\alpha_1\alpha_2$. When we deal with sequences of sequences, for clarity we sometimes put an element of a sequence that is itself a sequence into brackets $\langle\cdot\rangle$. For example, $\alpha_1 \langle\beta\rangle \alpha_2$ denotes a sequence containing another sequence β as one of its elements. We write $g[x : y]$ for the function that has the same value as g everywhere, except for x, where it has the value y. We write $\_$ for an expression whose value is irrelevant and implicitly existentially quantified. We denote the powerset of a set X with $\mathcal{P}(X)$, and the disjoint union of sets with $\uplus$.

Programming Language. We consider a machine with n CPUs, indexed by $\mathsf{CPUid} = \{1, \ldots, n\}$ and a shared memory. The machine executes programs of the following form:

$$L ::= \{m = C_m \mid m \in M\} \qquad C(L) ::= \mathsf{let}\ L\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n$$

A program consists of a declaration of a library L, implementing a set of methods $M \subseteq \mathsf{Method}$, and its client, specifying a command C_t to be run by the (hardware) thread in each CPU t. For the above program we let $\mathsf{sig}(L) = M$. To simplify presentation, we assume that the program is stored separately from the memory.

It is technically convenient for us to abstract from a particular syntax of thread and method bodies C_t and C_m and represent them using *control-flow graphs*. Namely, assume a set of primitive commands PComm (defined below). A control-flow graph (CFG) over the set PComm is a tuple $(N, T, \mathsf{start}, \mathsf{end})$, consisting of the set of program positions N, the control-flow relation $T \subseteq N \times \mathsf{PComm} \times N$, and the initial and final positions $\mathsf{start}, \mathsf{end} \in N$. The edges of the CFG are annotated with primitive commands from PComm.

We represent a program $C(L)$ by a collection of CFGs: the client command C_t for a CPU t is represented by $(N_t, T_t, \mathsf{start}_t, \mathsf{end}_t)$, and the body C_m of a method m by $(N_m, T_m, \mathsf{start}_m, \mathsf{end}_m)$. We often view this collection of CFGs for $C(L)$ as a single graph consisting of the node set $N = \biguplus_{t=1}^{n} N_t \uplus \biguplus_{m \in \mathsf{sig}(L)} N_m$ and the edge set $T = \biguplus_{t=1}^{n} T_t \uplus \biguplus_{m \in \mathsf{sig}(L)} T_m$.

Machine Configurations. The set of possible configurations Config of our machine is defined in Figure 1. The special configuration $\top$ results from the machine executing an illegal instruction, such as dereferencing a non-existent memory location. An ordinary configuration $(\mathsf{pc}, \theta, b, h, K) \in \mathsf{Config}$ consists of several components. The first one $\mathsf{pc} \in \mathsf{CPUid} \to \mathsf{Pos}$ gives the current instruction pointer of every CPU. When a CPU executes client code, its instruction pointer defines the program position of the client command being executed. Otherwise, it is given by a pair whose first component is the program position of the current library command, and the second one is the client position to return to when the library method finishes executing (one return position is sufficient, since, as explained below, we disallow nested method calls).

$$\mathsf{Loc} = \mathbb{N} \qquad \mathsf{Val} = \mathbb{Z} \qquad \mathsf{Heap} = \mathsf{Loc} \rightharpoonup_{fin} \mathsf{Val}$$
$$\mathsf{Pos} = N \uplus (N \times N) \qquad \mathsf{Reg} = \{\mathtt{r}_1, \ldots, \mathtt{r}_m\} \qquad \mathsf{RegBank} = \mathsf{Reg} \to \mathsf{Val}$$
$$\mathsf{Buff} = ((\mathsf{Loc} \times \mathsf{Val})^+ \cup \{\mathsf{lock}, \mathsf{call}, \mathsf{ret}\})^*$$
$$\mathsf{Config} = \{\top\} \cup ((\mathsf{CPUid} \to \mathsf{Pos}) \times (\mathsf{CPUid} \to \mathsf{RegBank}) \times (\mathsf{CPUid} \to \mathsf{Buff}) \times \mathsf{Heap} \times \mathcal{P}(\mathsf{CPUid}))$$

Fig. 1. The set of machine configurations

Each CPU in the machine has a set of registers Reg, whose values are defined by $\theta \in \mathsf{CPUid} \to \mathsf{RegBank}$. The machine memory $h \in \mathsf{Heap}$ is represented as a finite partial function from existing memory locations to the values they store. The component $K \in \mathcal{P}(\mathsf{CPUid})$ defines the set of *active* CPUs that can currently execute a command and is used to implement atomic execution of certain commands.

The component $b \in \mathsf{CPUid} \to \mathsf{Buff}$ describes the state of all store buffers in the machine, each represented by a sequence of write requests with newest coming first. The contents of store buffers in our configurations differ from those prescribed by the TSO memory model [15] in two ways.

First, in TSO every entry in a store buffer is represented by a single location-value pair, whereas we use a sequence of those. In our semantics, all the locations in such a sequence are written to the memory atomically. This functionality is not provided by the hardware; we use it for expressing the semantics of library specifications, which might include atomic blocks performing several writes (see the seqlock example in Section 4).

Second, to formulate linearizability, we need to maintain some auxiliary information about executions, recorded by call, ret and lock entries in a store buffer. The marker lock is used to implement atomic commands performing several writes to different locations in memory. The markers call and ret get added to the buffer upon a call to or a return from the library, respectively, and thus delimit entries added by library method invocations and client code. They are used to generate additional actions in histories of interactions between the client and the library needed to define linearizability on TSO. We note that, despite store buffers in our configurations including call and ret markers, the semantics we define below corresponds to the standard TSO one, in the sense that erasing the markers from store buffers in all configurations of a given execution yields a valid execution in the standard TSO semantics.

Primitive Commands. The set of primitive commands is defined as follows:

$$\mathsf{PComm} = \mathsf{Local} \uplus \mathsf{Read} \uplus \mathsf{Write} \uplus \{m \mid m \in \mathsf{Method}\} \uplus \{\mathsf{lock}, \mathsf{unlock}, \mathsf{xlock}, \mathsf{xunlock}\}.$$

Here Local, Read and Write are unspecified sets of commands such that:

- commands in Local access only CPU registers;
- commands in Read read a single location in memory and write its contents into the register $\mathtt{r}_1$;
- commands in Write write to a single location in memory.

We also have library method calls and the commands lock and unlock that lock the machine, allowing several commands to be executed atomically, and unlock it. We assume that parameters and return values of methods are passed via CPU registers. If a

client needs to preserve register values when calling a library method, it can save them in memory before the call and restore them when the method returns. The xlock and xunlock commands act as lock and unlock, except they have a built-in memory barrier, flushing the store buffer of the CPU executing the command. We call a sequence of commands bracketed by lock and unlock, or xlock and xunlock, an *atomic block*.

For every command $c \in \mathsf{Local} \uplus \mathsf{Read} \uplus \mathsf{Write}$, we assume a transformer:

- $f_c : \mathsf{RegBank} \to \mathcal{P}(\mathsf{RegBank})$ for $c \in \mathsf{Local}$ defining how the command changes the registers of the CPU executing it;
- $f_c : \mathsf{RegBank} \to \mathcal{P}(\mathsf{Loc})$ for $c \in \mathsf{Read}$ defining the location read;
- $f_c : \mathsf{RegBank} \to \mathcal{P}(\mathsf{Loc} \times \mathsf{Val})$ for $c \in \mathsf{Write}$ defining the location and the value written.

Note that we allow the execution of primitive commands to be non-deterministic. As in this paper we are dealing with low-level programs, we do not assume a built-in allocator, and thus do not consider commands for memory (de)allocation as primitive.

We place certain restrictions on CFGs over the above set PComm. Namely, we assume that on any path in a CFG, (x)lock and (x)unlock commands alternate correctly. In particular, we disallow nested (x)lock instructions. We assume that every method called in the program is defined, and we disallow nested method calls as well as method calls inside atomic blocks.

Let E, F denote expressions over the set of registers Reg, and $[\![E]\!]r$ the result of evaluating the expression E in the register bank r. Then we can define sample primitive commands

$$\mathsf{havoc} \in \mathsf{Local},\ \ \mathsf{assume}(E) \in \mathsf{Local},\ \ \mathsf{read}(E) \in \mathsf{Read},\ \ \mathsf{write}(E,F) \in \mathsf{Write}$$

with the following semantics:

$$f_{\mathsf{havoc}}(r) = \mathsf{RegBank}; \qquad f_{\mathsf{assume}(E)}(r) = \{r\}, \ \text{if } [\![E]\!]r \neq 0;$$
$$f_{\mathsf{read}(E)}(r) = \{[\![E]\!]r\}; \qquad f_{\mathsf{assume}(E)}(r) = \emptyset, \ \text{if } [\![E]\!]r = 0;$$
$$f_{\mathsf{write}(E,F)}(r) = \{([\![E]\!]r, [\![F]\!]r)\}.$$

The read and write commands have the expected meaning. The havoc command assigns arbitrary values to all registers. The assume(E) command acts as a filter on states, choosing only those where E evaluates to non-zero values. Using assume(E), a conditional branch on the value of E can be implemented with the CFG edges $(v, \mathsf{assume}(E), v_1)$ and $(v, \mathsf{assume}(!E), v_2)$, where $!E$ denotes the C-style negation.

Given the above commands, a memory barrier can be implemented as "xlock; xunlock". We can also implement the well-known atomic compare-and-swap (CAS) operation. A CAS takes three arguments: a memory address `addr`, an expected value `v1` and a new value `v2`. It atomically reads the memory address and updates it with the new value when the address contains the expected value; otherwise, it does nothing. In our language, we define `CAS(addr, v1, v2)` as syntactic sugar for the control-flow graph representation of:

```
xlock;
if (*addr == v1) { *addr = v2; xunlock; return 1; }
else { xunlock; return 0; }
```

Actions and Traces. Transitions in our operational semantics are labelled using *actions* of the form

$$\begin{aligned}\varphi \in \mathsf{Act} ::= {} & (t, \mathsf{read}(x,u)) \mid (t, \mathsf{write}(x,u)) \mid (t, \mathsf{flush}(x,u)) \mid (t, \mathsf{flush}(\mathsf{call})) \mid \\ & (t, \mathsf{flush}(\mathsf{ret})) \mid (t, \mathsf{lock}) \mid (t, \mathsf{unlock}) \mid (t, \mathsf{xlock}) \mid (t, \mathsf{xunlock}) \mid \\ & (t, \mathsf{call}\ m(r)) \mid (t, \mathsf{ret}\ m(r))\end{aligned}$$

where $t \in \mathsf{CPUid}$, $x \in \mathsf{Loc}$, $u \in \mathsf{Val}$, $m \in \mathsf{Method}$ and $r \in \mathsf{RegBank}$. Here $(t, \mathsf{write}(x,u))$ corresponds to enqueuing a pending write of u to the location x into the store buffer of CPU t, $(t, \mathsf{flush}(x,u))$ to flushing a pending write of u to the location x from the store buffer of t into the shared memory, $(t, \mathsf{flush}(\mathsf{call}))$ or $(t, \mathsf{flush}(\mathsf{ret}))$ to discarding a call or ret marker from the head of a store buffer. The last two actions record moments of time when entries in a store buffer written by a given library method invocation start to get flushed and are flushed completely, which are needed in the formulation of linearizability as we explained in Section 1. The rest of the actions have the expected meaning. Since parameters and return values of library methods are passed via CPU registers, we record their values in call and return actions.

We call a (finite or infinite) sequence of actions a *trace* and adopt the standard notation: $\lambda(i)$ is the i-th action in the trace λ, $|\lambda|$ is the length of the trace λ ($|\lambda| = \omega$ if λ is infinite), and $\lambda|_t$ is the projection of λ to actions by CPU t.

Program Semantics. The operational semantics of a program $C(L)$ is defined by the transition relation $\longrightarrow_{C(L)}$: $\mathsf{Config} \times \mathsf{Act}^* \times \mathsf{Config}$ in Figure 2. We remind the reader that T in the figure is the control-flow relation of $C(L)$. To handle transitions inside the library code, we lift it to program positions $N \uplus (N \times N)$ as follows:

$$\hat{T} = T \cup \{((v, v_0), c, (v', v_0)) \mid (v, c, v') \in T \wedge v_0 \in N\}.$$

The LOCAL rule handles the execution of commands that access registers only. These and other commands can only be executed by a CPU t if it is included into the set of active CPUs, represented by the last component of a configuration.

A write by a CPU to a location in memory does not happen immediately; instead, a pair of the location and the value to be written is added to the tail of the corresponding store buffer (WRITE). Recall that the newest entry comes first in the store buffer. When the location being written does not exist, the write command faults (WRITE-$\top$).

The READ rule uses $\mathsf{lookup}(\alpha, h, x)$ to find the value stored for the address x in the store buffer α of the CPU executing the command or the memory h:

$$\mathsf{lookup}(\alpha, h, x) = \begin{cases} u, & \text{if } \alpha = \alpha_1 \langle \beta_1 (x,u) \beta_2 \rangle \alpha_2 \text{ and} \\ & \alpha_1, \beta_1 \text{ do not contain entries for } x; \\ h(x), & \text{if } x \in \mathsf{dom}(h) \text{ and } \alpha \text{ does not contain entries for } x; \\ \top, & \text{otherwise.} \end{cases}$$

If there are entries for x in the store buffer, the read takes the value in the newest one; otherwise, it looks up the value in memory. If the location being read does not exist, lookup returns $\top$. According to READ, the value read is stored in the register $\mathtt{r}_1$.

$$\frac{t \in K \qquad (\rho, c, \rho') \in \hat{T} \qquad c \in \mathsf{Local} \qquad r' \in f_c(r)}{\mathsf{pc}[t:\rho], \theta[t:r], b, h, K \xrightarrow{\varepsilon}_{C(L)} \mathsf{pc}[t:\rho'], \theta[t:r'], b, h, K} \text{ LOCAL}$$

$$\frac{(\rho, c, \rho') \in \hat{T} \qquad c \in \mathsf{Write} \qquad (x,u) \in f_c(r) \qquad x \in \mathsf{dom}(h)}{\mathsf{pc}[t:\rho], \theta[t:r], b[t:\alpha], h, K \xrightarrow{(t,\mathsf{write}(x,u))}_{C(L)} \mathsf{pc}[t:\rho'], \theta[t:r], b[t:(x,u)\,\alpha], h, K} \text{ WRITE}$$

$$\frac{t \in K \qquad (\rho, c, \rho') \in \hat{T} \qquad c \in \mathsf{Write} \qquad (x,u) \in f_c(r) \qquad x \notin \mathsf{dom}(h)}{\mathsf{pc}[t:\rho], \theta[t:r], b, h, K \xrightarrow{\varepsilon}_{C(L)} \top} \text{ WRITE-}\top$$

$$\frac{t \in K \qquad (\rho, c, \rho') \in \hat{T} \qquad c \in \mathsf{Read} \qquad x \in f_c(r) \qquad u = \mathsf{lookup}(\alpha, h, x) \neq \top}{\mathsf{pc}[t:\rho], \theta[t:r], b[t:\alpha], h, K \xrightarrow{(t,\mathsf{read}(x,u))}_{C(L)} \mathsf{pc}[t:\rho'], \theta[t:r[\mathtt{r}_1:u]], b[t:\alpha], h, K} \text{ READ}$$

$$\frac{t \in K \qquad (\rho, c, \rho') \in \hat{T} \qquad c \in \mathsf{Read} \qquad x \in f_c(r) \qquad \mathsf{lookup}(\alpha, h, x) = \top}{\mathsf{pc}[t:\rho], \theta[t:r], b[t:\alpha], h, K \xrightarrow{\varepsilon}_{C(L)} \top} \text{ READ-}\top$$

$$\frac{(\rho, \mathsf{lock}, \rho') \in \hat{T}}{\mathsf{pc}[t:\rho], \theta, b[t:\alpha], h, \mathsf{CPUid} \xrightarrow{(t,\mathsf{lock})}_{C(L)} \mathsf{pc}[t:\rho'], \theta, b[t:\mathsf{lock}\,\alpha], h, \{t\}} \text{ LOCK}$$

$$\frac{(\rho, \mathsf{unlock}, \rho') \in \hat{T}}{\mathsf{pc}[t:\rho], \theta, b[t:(x_1,u_1)\ldots(x_l,u_l)\,\mathsf{lock}\,\alpha], h, \{t\} \xrightarrow{(t,\mathsf{unlock})}_{C(L)} \mathsf{pc}[t:\rho'], \theta, b[t:\langle(x_1,u_1)\ldots(x_l,u_l)\rangle\,\alpha], h, \mathsf{CPUid}} \text{ UNLOCK}$$

$$\frac{}{\mathsf{pc}, \theta, b[t:\alpha\,\langle(x_1,u_1)\ldots(x_l,u_l)\rangle], h, \mathsf{CPUid} \xrightarrow{(t,\mathsf{flush}(x_l,u_l))\ldots(t,\mathsf{flush}(x_1,u_1))}_{C(L)} \mathsf{pc}, \theta, b[t:\alpha], h[x_l:u_l]\ldots[x_1:u_1], \mathsf{CPUid}} \text{ FLUSH}$$

$$\frac{\beta \in \{\mathsf{call}, \mathsf{ret}\}}{\mathsf{pc}, \theta, b[t:\alpha\beta], h, \mathsf{CPUid} \xrightarrow{(t,\mathsf{flush}(\beta))}_{C(L)} \mathsf{pc}, \theta, b[t:\alpha], h, \mathsf{CPUid}} \text{ FLUSH-MARKER}$$

$$\frac{(\rho, \mathsf{xlock}, \rho') \in \hat{T}}{\mathsf{pc}[t:\rho], \theta, b[t:\varepsilon], h, \mathsf{CPUid} \xrightarrow{(t,\mathsf{xlock})}_{C(L)} \mathsf{pc}[t:\rho'], \theta, b[t:\varepsilon], h, \{t\}} \text{ XLOCK}$$

$$\frac{(\rho, \mathsf{xunlock}, \rho') \in \hat{T}}{\mathsf{pc}[t:\rho], \theta, b[t:(x_1,u_1)\ldots(x_l,u_l)], h, \{t\} \xrightarrow{(t,\mathsf{flush}(x_l,u_l))\ldots(t,\mathsf{flush}(x_1,u_1))(t,\mathsf{xunlock})}_{C(L)} \mathsf{pc}[t:\rho'], \theta, b[t:\varepsilon], h[x_l:u_l]\ldots[x_1:u_1], \mathsf{CPUid}} \text{ XUNLOCK}$$

$$\frac{(v, m, v') \in T}{\mathsf{pc}[t:v], \theta[t:r], b[t:\alpha], h, \mathsf{CPUid} \xrightarrow{(t,\mathsf{call}\ m(r))}_{C(L)} \mathsf{pc}[t:(\mathsf{start}_m, v')], \theta[t:r], b[t:\mathsf{call}\,\alpha], h, \mathsf{CPUid}} \text{ CALL}$$

$$\frac{}{\mathsf{pc}[t:(\mathsf{end}_m, v')], \theta[t:r], b[t:\alpha], h, \mathsf{CPUid} \xrightarrow{(t,\mathsf{ret}\ m(r))}_{C(L)} \mathsf{pc}[t:v'], \theta[t:r], b[t:\mathsf{ret}\,\alpha], h, \mathsf{CPUid}} \text{ RET}$$

Fig. 2. Operational TSO semantics

A CPU executing lock makes itself the only active CPU, preventing the others from executing commands[1] (LOCK). The commands executed within the corresponding atomic block, i.e., until the CPU calls unlock (UNLOCK) are thus not interleaved with commands of other CPUs. A lock command also adds a lock marker to the tail of the store buffer, thus delimiting the write requests issued within the atomic block. The corresponding unlock command then uses the lock marker to gather these write requests into a single buffer entry. Since we prohibit method calls inside atomic blocks, this entry does not contain call or ret markers.

A CPU may at any point decide to flush the entry at the head of the store buffer into memory (FLUSH). All the writes in the entry are flushed at the same time, thus ensuring that writes made in an atomic block take effect atomically. A CPU can also discard the marker at the head of the store buffer (FLUSH-MARKER). Although this does not modify the memory, we use the corresponding action, recorded in the transition relation, to formulate linearizability (Section 4). For technical reasons, it is convenient for us to prohibit flushes inside an atomic block delimited by lock and unlock. Thus, the FLUSH and FLUSH-MARKER require the set of active CPUs to be CPUid.

The xlock command (XLOCK) can only be executed when the store buffer is empty and thus forces the CPU to flush its store buffer beforehand using FLUSH and FLUSH-MARKER. For this reason, it does not need to insert a lock marker into the buffer: by the end of the atomic block the buffer will only contain writes issued inside it. The xunlock command flushes all these entries into the memory (XUNLOCK).

The rules CALL and RET handle calls to and returns from methods. Upon a method call, the return point is saved as a component in the new thread position, a call marker is added to the tail of the store buffer, and the method starts executing from the corresponding starting node of its CFG. Upon a return, the return point is read from the current program position, and a ret marker is added to the tail of the store buffer. Note that configurations in CALL and RET rules have CPUid as the set of active CPUs, since we prohibit method calls inside atomic blocks.

We note that the store buffers arising in executions of $C(L)$ as defined in Figure 2 are not arbitrary elements of Buff, but satisfy certain properties: e.g., call and ret markers in them alternate correctly, and they contain at most one lock marker. We formalise such properties in [5, Appendix A].

Implementations of the TSO memory model usually guarantee that store buffers are fair, in the sense that, eventually, every write request in a buffer will be flushed into the memory. Our results can be extended to accommodate this constraint; however, we do not handle it in this paper so as not to obfuscate presentation.

A *computation* of $C(L)$ is a sequence of transitions using $\longrightarrow_{C(L)}$. For a computation τ, we let $\mathsf{trace}(\tau)$ be the trace obtained by concatenating all the annotations of transitions in τ. In the following, we assume that program properties of interest are linear-time properties over sets of program traces. We denote with $\xrightarrow{\lambda}{}^*_{C(L)}$ the reflexive and transitive closure of $\longrightarrow_{C(L)}$, where λ is obtained by concatenating the transition annotations.

[1] The semantics of TSO [15] locks only the memory bus in this case, which allows other CPUs to execute local commands affecting only their registers. For simplicity, we chose to disallow all commands.

Let $I \subseteq \mathsf{Heap}$ be the set of initial heaps that the program $C(L)$ expects to execute from. We define the set of its initial configurations as

$$\Sigma_0(I) = \{(\mathsf{pc}_0, \theta_0, b_0, h_0, \mathsf{CPUid}) \mid \forall t \in \mathsf{CPUid}.\, \mathsf{pc}_0(t) = \mathsf{start}_t \wedge b_0(t) = \varepsilon \wedge h_0 \in I\}.$$

We define the semantics $[\![C(L)]\!]I$ of $C(L)$ executing from I as the set of computations with initial configurations from $\Sigma_0(I)$. We say that the program $C(L)$ is *safe* for I, if it is not the case that $\sigma_0 \xrightarrow{\lambda}{}^*_{C(L)} \top$ for some λ and $\sigma_0 \in \Sigma_0(I)$. Informally, a program is safe when it accesses only allocated memory. Safety can be established using existing logics for reasoning about programs running on TSO [16,20].

3 Library-Local and Client-Local Semantics

Consider a library L and a program $C(L)$ using this library:

$$L \;=\; \{m = C_m \mid m \in M\}, \qquad\qquad C(L) \;=\; \mathsf{let}\ L\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n.$$

To formulate the definition of linearizability and the Abstraction Theorem, we need to give a semantics to parts of $C(L)$: the library L considered in isolation from its client and the client C considered in isolation from the implementation of the library it uses. In this section, we specialise the semantics of programs in Section 2 to such *library-local* and *client-local* semantics describing all possible behaviours of the corresponding components.

Let us lift the operation of the disjoint union of heaps to sets of heaps pointwise:

$$\forall I_1, I_2 \subseteq \mathsf{Heap}.\, I_1 \circ I_2 = \{h_1 \uplus h_2 \mid h_1 \in I_1 \wedge h_2 \in I_2\}.$$

We assume that the set I of initial heaps of $C(L)$ satisfies $I = I_c \circ I_l$ for some $I_c, I_l \subseteq \mathsf{Heap}$ such that for any $h_c \in I_c$ and $h_l \in I_l$, $h_c \uplus h_l$ is defined. Here I_c and I_l are meant to represent parts of initial heaps used by the client C and the library L, respectively; the initial heaps of $C(L)$ are obtained as the $\circ$-combination of these.

Recall that n is the number of CPUs in our machine. To give a library-local semantics to L, we consider the program $\mathsf{MGC}(L) \;=\; \mathsf{let}\ L\ \mathsf{in}\ C_1^{\mathsf{mgc}} \parallel \ldots \parallel C_n^{\mathsf{mgc}}$, where C_t^{mgc} has the CFG

$$(\{v^t_{\mathsf{mgc}}\}, \{(v^t_{\mathsf{mgc}}, \mathsf{havoc}, v^t_{\mathsf{mgc}}), (v^t_{\mathsf{mgc}}, m, v^t_{\mathsf{mgc}}) \mid m \in \mathsf{sig}(L)\}, v^t_{\mathsf{mgc}}, v^t_{\mathsf{mgc}}).$$

The program $\mathsf{MGC}(L)$ is the *most general client* of the library L, whose hardware threads on every CPU repeatedly invoke library methods in any order and with any parameters possible. The latter are passed via registers, set arbitrarily by the havoc command. The set of computations $[\![\mathsf{MGC}(L)]\!]I_l$ thus includes all library behaviours under any possible client (this fact is formalised in Lemma 6, Section 5).

In practice, a library often tolerates only calls from clients adhering to a certain policy. For example, a spinlock implementation might expect client calls to `acquire` and `release` methods to alternate. We can take this into account by restricting the most general client appropriately. While libraries in our examples do rely on the client satisfying such constraints, to simplify presentation we do not formalise them here.

To define the client-local semantics of the client C, we consider the program

$$C_M(\cdot) \;=\; \mathsf{let}\,\{m = C_m^{\mathsf{stub}} \mid m \in M\}\,\mathsf{in}\,C_1 \parallel \ldots \parallel C_n$$

where the body C_m^{stub} of every method m has the CFG $(\{v_{\mathsf{start}}^m\}, \{(v_{\mathsf{start}}^m, \mathsf{havoc}, v_{\mathsf{end}}^m)\}, v_{\mathsf{start}}^m, v_{\mathsf{end}}^m)$. That is, every method in $C_M(\cdot)$ is implemented by a stub that returns immediately after having been called, scrambling all the registers. Since return values of library methods are stored in registers, the set of computations $[\![C_M(\cdot)]\!]I_c$ generates all executions of the client assuming any behaviour of the library it uses.

Note that both library-local and client-local semantics allow store buffer entries of the corresponding component to be flushed non-deterministically while the other component is running, since this is possible in the semantics of the whole program. Similarly, we add call and ret markers to the store buffer when calling a method stub in the client-local and library-local semantics.

We say that a client C, respectively, a library L is safe for I_c, respectively, I_l, if so is $C_M(\cdot)$, respectively, $\mathsf{MGC}(L)$ (see Section 2). As we have noted before, the safety of a library or a client can be established using logics for TSO [16,20]. Note that in the client-local or the library-local semantics, the program runs on the state owned by the corresponding component and faults when accessing memory locations not belonging to it. Thus, the safety of the client and the library ensures that they cannot corrupt each other's state. We rely crucially on this in establishing the Abstraction Theorem for the notion of linearizability we propose. It can also be shown that, when the client C and the library L are safe, so is the complete program $C(L)$ (Lemma 6, Section 5).

4 Linearizability on TSO

When defining linearizability, we are not interested in internal steps recorded in library computations, but only in the interactions of the library with its client. We record such interactions using *histories*, which are traces including only actions from the following subset of Act:

$$\mathsf{HAct} \;::=\; (t, \mathsf{call}\; m(r)) \mid (t, \mathsf{ret}\; m(r)) \mid (t, \mathsf{flush}(\mathsf{call})) \mid (t, \mathsf{flush}(\mathsf{ret}))$$

where $t \in \mathsf{CPUid}$, $m \in \mathsf{Method}$, $r \in \mathsf{RegBank}$. Recall that here r records the values of registers of the CPU that calls a library method or returns from it, which serve as parameters or return values. We define the history $\mathsf{history}(\tau)$ corresponding to a computation τ of the program $C(L)$ by projecting $\mathsf{trace}(\tau)$ to actions from HAct.

In contrast to histories used in the classical definition of linearizability [11], ours include two new types of actions needed for defining linearizability on TSO: $(t, \mathsf{flush}(\mathsf{call}))$ and $(t, \mathsf{flush}(\mathsf{ret}))$, denoting times when the CPU t flushes a call or a ret marker from its store buffer. We first formulate our definition, and then explain the motivation behind it.

Definition 1. *The* ***linearizability relation*** *is a binary relation $\sqsubseteq$ on histories defined as follows: $H \sqsubseteq H'$ if $\forall t \in \mathsf{CPUid}.\, H|_t = H'|_t$ and there is a bijection $\pi\colon \{1, \ldots, |H|\} \to \{1, \ldots, |H'|\}$ such that $\forall i.\, H(i) = H'(\pi(i))$ and*

$$\begin{aligned}(i < j \wedge (H(i) = (\_, \mathsf{ret}\ \_) \vee H(i) = (\_, \mathsf{flush}(\mathsf{ret})))\\ \wedge\, (H'(j) = (\_, \mathsf{call}\ \_) \vee H'(j) = (\_, \mathsf{flush}(\mathsf{call})))) \Rightarrow \pi(i) < \pi(j).\end{aligned}$$

That is, a history H' linearizes a history H when it is a permutation of the latter preserving the order of certain types of actions. We lift the notion of linearizability to libraries using the library-local semantics of Section 3.

Definition 2. *For libraries L_1 and L_2 safe for I_l and such that* $\mathsf{sig}(L_1) = \mathsf{sig}(L_2)$*, we say that L_2* ***linearizes*** *L_1, written $L_1 \sqsubseteq L_2$, if*

$$\forall H_1 \in \mathsf{history}([\![\mathsf{MGC}(L_1)]\!] I_l).\ \exists H_2 \in \mathsf{history}([\![\mathsf{MGC}(L_2)]\!] I_l).\ H_1 \sqsubseteq H_2.$$

Thus, L_2 linearizes L_1 if every behaviour of the latter under the most general client may be reproduced in a linearized form by the former.

Discussion. A good definition of linearizability has to allow replacing a library implementation with its specification while keeping client behaviours reproducible (as formalised by the Abstraction Theorem in Section 5). However, linearizability itself is defined between libraries considered in isolation from their clients. In Definition 2, this is achieved by considering executions of libraries under their most general clients (Section 3), which can only refer to store buffer entries inserted by write commands in library code. When a library is used by a client, the store buffer mixes entries inserted by the two components. As we noted in Section 1, in this case the library can affect the client via the store buffer, e.g., by executing a memory barrier or leaving an unflushed entry blocking newer client entries from being flushed. The $(\_, \mathsf{flush}(\mathsf{call}))$ and $(\_, \mathsf{flush}(\mathsf{ret}))$ actions in histories record the necessary information about library behaviour of this kind, as we now explain.

Recall the analogy from Section 1, where we viewed the contents of a store buffer as a sandwich consisting of blocks of entries inserted there by an invocation of a library method or a fragment of client computation between two such invocations. The call and ret markers delimit the layers in this sandwich. For example, at some point in an execution of $C(L)$, the store buffer of some CPU might have the following contents:

$$\mathsf{ret}\,(x_5, u_5)\,\mathsf{call}\,(x_4, u_4)\,\mathsf{ret}\,(x_3, u_3)\,(x_2, u_2)\,\mathsf{call}\,(x_1, u_1), \tag{1}$$

where the leftmost end contains the newest entry. From the call and ret markers, we can immediately conclude that the write to x_1 was inserted by the client before calling a library method, the writes to x_2 and x_3 were by the library method invocation, the write to x_4 was again by the client, and the write to x_5 was by the next method invocation on this CPU.

The most general client exercises the library methods under all possible input parameters, but does not perform writes by itself. For this reason, a store buffer in the most general client of a library never has entries between a call marker and an older ret marker (we formalise this in [5, Appendix A]). For example, a computation of the most general client of the library with the same library method invocations as in the one producing (1) might have the store buffer

$$\mathsf{ret}\,(x_5, u_5)\,\mathsf{call}\,\mathsf{ret}\,(x_3, u_3)\,(x_2, u_2)\,\mathsf{call}, \tag{2}$$

which contains only library entries from (1). Thus, when considering a library in isolation from its client in defining linearizability, the call and ret markers let us determine

the places in the store buffer where client entries might be located in a corresponding execution of a complete program.

Consider an execution of the most general client of a library in which the CPU flushes a library entry (e.g., (x_3, u_3) in (2)). Since store buffers are FIFO, in the corresponding execution of a particular client with the same library behaviour, this will *assume* that the client entries in the store buffer older than it have been flushed (e.g., (x_1, u_1) in (1)). Conversely, flushing a library entry (e.g., (x_3, u_3) in (1)) preceding a client one (e.g., (x_4, u_4) in (1)) will *guarantee* that the client entry can now be flushed. For the Abstraction Theorem to hold, in Definition 2 we need to make sure that the executions of the most general clients producing histories H_1 and H_2 make the same assumptions and give the same guarantees concerning times when client entries are flushed. This is the reason for including flushes of call and ret markers into histories. The position of a $(t, \mathsf{flush}(\mathsf{call}))$ action in a history produced by the most general client defines a moment of time by which, in a complete program, all older client writes in the store buffer of t *must* be flushed for the library to be able to flush the entries from the layer following the call marker. The position of a $(t, \mathsf{flush}(\mathsf{ret}))$ action defines a moment starting from which the client entries from the layer following the ret marker *may* be flushed. In our definition of linearizability, we require that the two histories considered have the same history actions describing how store buffers are modified during the execution. Hence, in two executions corresponding to the histories, libraries make the same assumptions and give the same guarantees concerning the use of store buffers.

Like the classical definition of linearizability, ours requires preserving the order between non-overlapping library method invocations; two invocations do not overlap in a history if the return of one precedes the call of the other. This is needed for the Abstraction Theorem to hold, since the client code executed in between two non-overlapping method invocations can notice their order. To handle TSO correctly, our definition also takes into account intervals during which all the writes of a library method invocation were being flushed: it requires preserving the order between two such non-overlapping intervals or non-overlapping interval of this kind and a library method invocation. This is expressed by preserving the order of $(\_, \mathsf{flush}(\mathsf{ret}))$ preceding $(\_, \mathsf{flush}(\mathsf{call}))$, $(\_, \mathsf{flush}(\mathsf{ret}))$ preceding $(\_, \mathsf{call}\_)$, and $(\_, \mathsf{ret}\_)$ preceding $(\_, \mathsf{flush}(\mathsf{call}))$. The requirement is again needed to validate the Abstraction Theorem.

We note that our definition of linearizability is flexible in the following sense: it puts restrictions on times when call and ret markers are flushed, but not on how many ordinary entries a given method invocation inserts into the store buffer. For example, this allows us to relate a library implementation writing to some part of the memory accessed only by a given CPU to its specification that does not write to any local state.

Example. Even though we formalise our results for programs represented by their CFGs, for readability in our examples we use a C-like language. Its programs can be translated to CFGs in the standard way. We assume that global variables are allocated at fixed addresses in memory, and local variables are stored in CPU registers.

Figure 3 presents a simplified version of a seqlock [3]—an efficient implementation of a readers-writer protocol based on version counters used in the Linux kernel. Two memory addresses `x1` and `x2` make up a conceptual register that a single hardware thread can write to, and any number of other threads can attempt to read from. A version

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

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

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

Fig. 3. Seqlock implementation L_{seqlock}

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

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

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

Fig. 4. Seqlock specification $L^{\sharp}_{\mathsf{seqlock}}$. Here `nondet()` represents a non-deterministic choice.

number is stored at `c`. The writing thread maintains the invariant that the version number is odd during writing by incrementing it before the start of and after the finish of writing. A reader checks that the version number is even before attempting to read (otherwise it could see an inconsistent result by reading while `x1` and `x2` are being written). After reading, the reader checks that the version has not changed, thereby ensuring that no write has overlapped the read. Note that neither the `write` nor the `read` operation includes a memory barrier, which means that writes to `x1`, `x2` and `c` may not be visible to readers immediately.

We give a specification to seqlock using the abstract implementation in Figure 4. Instead of using a version counter, this implementation just locks the machine while reading from or writing to `x1` and `x2`. According to the semantics of Section 2, the writes to `x1` and `x2` performed by `write` are stored in a single entry of the corresponding store buffer and are written to the shared memory atomically. This specifies that the implementation of a seqlock indeed ensures the illusion of atomicity. However, we also need our specification to capture the effect of the library executing on a weak memory model—the fact that the writes to `x1` and `x2`, although executed atomically, may still be delayed due to the presence of store buffers. This is because the delay can be noticed by certain clients and can result in a non-SC behaviour. For example, using a seqlock, we can reproduce the example from Section 1 yielding non-SC behaviour as shown in Figure 4. To capture this, the specification of `write` ensures atomicity by a pair of lock and unlock commands, which do not flush the writes to the memory immediately.

Thus, we have two atomic actions associated with the abstract `write` method: one that writes to the store buffer and the other that flushes the writes to the memory, possibly after the method returns. This is different from the classical definition of linearizability on a sequentially consistent memory model [11], which requires methods in the specification to be implemented by one atomic action.

```
        x1 = x2 = y = 0;
write(1, 1);  ||  y = 1;
b = y;        ||  read(&a1, &a2);
```

$\{\texttt{a1} = \texttt{b2} = \texttt{b} = 0\}$

Fig. 5. A client of L_{seqlock} producing a non-SC behaviour

As the following theorem shows, the abstract implementation $L^{\sharp}_{\mathsf{seqlock}}$ in Figure 4 indeed linearizes the concrete one L_{seqlock} in Figure 3.

Theorem 3. $L_{\mathsf{seqlock}} \sqsubseteq L^{\sharp}_{\mathsf{seqlock}}$.

The proof is given in [5, Appendix A]; here we discuss it informally. The proof is similar to proofs of classical linearizability using linearization points [11], although here methods of the abstract implementation contain more than one atomic action. We consider the most general clients of the concrete and the abstract implementations of the library running alongside each other. For every execution of the client of the concrete library, we construct the corresponding execution of the client of the abstract one by firing transitions of the latter at certain times during the execution of the former.

For example, the abstract `read` method is executed when the corresponding concrete one reads `x2` for the last time. The code of the abstract `write` method is executed when the concrete one writes to `x2`. Finally, a store buffer entry containing writes to `x1` and `x2` by the abstract `write` method is flushed together with the second write to `c` by the corresponding concrete method invocation. To prove that this flush in the abstract implementation does not contradict the FIFO ordering of store buffers, we maintain an invariant relating the contents of the store buffers in the concrete and the abstract seqlock implementations.

Programs Producing Only SC Behaviours. By this time, the reader may wonder whether it is always necessary to expose the behaviour of a library with respect to store buffers in its specification. After all, many programs running on TSO only produce SC behaviours, and there are ways of effectively checking this [14,6,7]. Therefore, a valid question is whether we can use the usual definition of linearizability for libraries producing only SC behaviours when they are used by clients also behaving SC. Unfortunately, in general the answer is no. This is because, even if the most general client of a library $\mathsf{MGC}(L)$ and its client $C_{\mathsf{sig}(L)}(\cdot)$ only produce SC behaviours when considered in isolation, this may not be the case for the complete program $C(L)$ due to interactions of the two components via the store buffer. For example, the most general client of a single seqlock produces only SC behaviours, as it satisfies the triangular race freedom criterion of [14]. However, Figure 4 shows that if we use a seqlock *together with* a client that also happens to be SC by itself, we can get non-SC behaviours. This is not surprising: a seqlock is meant to ensure the atomicity of writes to and reads from a pair of locations, but it is not meant to make these reads and writes strongly consistent. Thus, the classical definition of linearizability is not sufficient to specify libraries even when constraining separate components of a program to behave SC.

5 Abstraction Theorem

We now justify that the notion of linearizability proposed in Section 4 is a correct one by establishing the Abstraction Theorem that allows abstracting an implementation of a library with its specification while reasoning about its client.

For a computation τ of $C(L)$ obtained from the semantics of Section 2, we denote with $\mathsf{client}(\tau)$ the projection of its trace $\lambda = \mathsf{trace}(\tau)$ to actions relevant to the client, i.e., executed by the client code or corresponding to flushes of client entries in store buffers. Formally, we include an action φ such that $\lambda = \lambda'\varphi\lambda''$ into the projection if:

- φ is included into $\mathsf{history}(\tau)$; or
- φ is not a flush action and is outside an invocation of a library method, i.e., it is not the case that $\lambda|_t = \lambda_1\,(t, \mathsf{call}\ \_)\,\lambda_2\varphi\lambda_3$, where λ_2 does not contain a $(t, \mathsf{ret}\ \_)$ action; or
- φ corresponds to a flush of a client entry in a store buffer, i.e., it is not the case that $\lambda|_t = \lambda_1\,(t, \mathsf{flush}(\mathsf{call}))\,\lambda_2\varphi\lambda_3$, where λ_2 does not contain a $(t, \mathsf{flush}(\mathsf{ret}))$ action.

We lift client to sets of computations pointwise.

The Abstraction Theorem states that the behaviour of a client of a concurrent library will stay reproducible on TSO if we replace the library by its abstract implementation related to the original one by our definition of linearizability.

Theorem 4 (Abstraction). *Consider $C(L_1)$ and $C(L_2)$ such that C is safe for I_c, L_1 and L_2 are safe for I_l and $L_1 \sqsubseteq L_2$. Then $C(L_1)$ and $C(L_2)$ are safe for $I = I_c \circ I_l$ and $\mathsf{client}(\llbracket C(L_1) \rrbracket I) \subseteq \mathsf{client}(\llbracket C(L_2) \rrbracket I)$.*

We provide a proof outline below and give the complete proof in [5, Appendix A]. The requirement that the client C be safe in the theorem is required to replace one library implementation with another: it ensures that C cannot access the internals of the library implementation.

From Theorem 4 it follows that, while reasoning about a client $C(L_1)$ of a library L_1, we can soundly replace L_1 with a simpler library L_2 linearizing L_1: if a linear-time property over client actions holds over $C(L_2)$, it will also hold over $C(L_1)$. Note that the abstract implementation is usually simpler than the original one (in most cases implemented using atomic blocks, like the one in Figure 4), which eases the proof of the resulting program. Thus, the proposed notion of linearizability and Theorem 4 enable compositional reasoning about programs running on TSO: they allow decomposing the verification of a whole program into the verification of its constituent components. We give an example of using this technique in Section 6.

The following corollary of Theorem 4, proved in [5, Appendix A], states that, like the classical notion of linearizability [11], ours is compositional: if several non-interacting libraries are linearizable, then so is their composition. Formally, consider libraries $L_1, \ldots, L_k$ with disjoint sets of declared methods and sets of initial heaps $I_1, \ldots, I_k$ such that

$$\forall \{i_1, \ldots, i_l\} \subseteq \{1, \ldots, k\}.\, \forall h_1 \in I_{i_1}, \ldots, h_l \in I_{i_l}.\, h_1 \uplus \ldots \uplus h_l \text{ is defined.}$$

We let the *composition* L of $L_1, \ldots, L_k$ be the library implementing all of their methods and having the set of initial heaps $I_1 \circ \ldots \circ I_k$.

Corollary 5 (Compositionality). *Consider libraries $L_1, \ldots, L_k$ and $L_1^\sharp, \ldots, L_k^\sharp$ such that L_j and $L_j^\sharp$ are safe for I_j, $j = 1..k$. Let L and $L^\sharp$ be the compositions of the respective sets of libraries. If $L_j \sqsubseteq L_j^\sharp$ for $j = 1..k$, then $L \sqsubseteq L^\sharp$.*

Proof Outline for Theorem 4. The proof of Theorem 4 relies on the following lemmas, proved in [5, Appendix A]. The first lemma shows that a computation of $C(L)$ generates two computations in the client-local and library-local semantics with the same history.

Lemma 6 (Decomposition). *If $C_{\mathsf{sig}(L)}(\cdot)$ and $\mathsf{MGC}(L)$ are safe for I_c and I_l, respectively, then $C(L)$ is safe for $I_c \circ I_l$ and*

$$\forall \tau \in [\![C(L)]\!](I_c \circ I_l).\, \exists \eta \in [\![C_{\mathsf{sig}(L)}(\cdot)]\!]I_c.\, \exists \xi \in [\![\mathsf{MGC}(L)]\!]I_l.\\ \mathsf{history}(\eta) = \mathsf{history}(\xi) \wedge \mathsf{client}(\tau) = \mathsf{client}(\eta).$$

The following lemma presents the core of the transformation used to convert a computation of $C(L_1)$ into one of $C(L_2)$ in Theorem 4: it shows that a computation of a most general client can be transformed into another of its computations with a given history linearized by the history of the original one.

Lemma 7 (Rearrangement). *Consider a library L safe for I_l and histories H, H' such that $H \sqsubseteq H'$. Then*

$$\forall \tau' \in [\![\mathsf{MGC}(L)]\!]I_l.\, \mathsf{history}(\tau') = H' \Rightarrow \exists \tau \in [\![\mathsf{MGC}(L)]\!]I_l.\, \mathsf{history}(\tau) = H.$$

Finally, the following lemma states that any pair of client-local and library-local computations agreeing on the history can be combined into a valid computation of $C(L)$.

Lemma 8 (Composition). *If $C_{\mathsf{sig}(L)}(\cdot)$ and $\mathsf{MGC}(L)$ are safe for I_c and I_l, respectively, then*

$$\forall \eta \in [\![C_{\mathsf{sig}(L)}(\cdot)]\!]I_c.\, \forall \xi \in [\![\mathsf{MGC}(L)]\!]I_l.\, \mathsf{history}(\eta) = \mathsf{history}(\xi) \Rightarrow\\ \exists \tau \in [\![C(L)]\!](I_c \circ I_l).\, \mathsf{client}(\tau) = \mathsf{client}(\eta).$$

Most of the proof of the Decomposition Lemma (Lemma 6) deals with maintaining a splitting of the state of $C(L)$ into the parts owned by the client and the library, including store buffer entries. The resulting partial states then define the computations of $C_{\mathsf{sig}(L)}(\cdot)$ and $\mathsf{MGC}(L)$. Conversely, the Composition Lemma (Lemma 8) composes the states of $C_{\mathsf{sig}(L)}(\cdot)$ and $\mathsf{MGC}(L)$ into states of $C(L)$ to construct an execution of the latter. The proof of the Rearrangement Lemma (Lemma 7) transforms τ' into τ by repeatedly permuting transitions in the computation according to a certain strategy to make its history equal to H.

Proof of Theorem 4. Lemma 6 implies that $C(L)$ is safe. We now need to transform a computation $\tau_1 \in [\![C(L_1)]\!]I$ of $C(L_1)$ into a computation $\tau_2 \in [\![C(L_2)]\!]I$ with the same client trace projection: $\mathsf{client}(\tau_1) = \mathsf{client}(\tau_2)$. To this end, we use the semantics

of Section 3, which defines the interpretation of L_1, L_2, $C_{\mathsf{sig}(L_1)}(\cdot)$ and their compositions. Namely, to transform τ_1 into τ_2, we first apply Lemma 6 to generate two computations from τ_1—a library-local computation $\xi_1 \in [\![\mathsf{MGC}(L_1)]\!] I_l$ and a client-local one $\eta \in [\![C_{\mathsf{sig}(L_1)}(\cdot)]\!] I_c$—such that $\mathsf{client}(\tau_1) = \mathsf{client}(\eta)$ and $\mathsf{history}(\tau_1) = \mathsf{history}(\eta) = \mathsf{history}(\xi_1)$. Note that the computation η of C thus constructed excludes the internal library actions. Since $L_1 \sqsubseteq L_2$, for some computation $\xi_2 \in [\![\mathsf{MGC}(L_2)]\!] I_l$, we have $\mathsf{history}(\xi_1) \sqsubseteq \mathsf{history}(\xi_2)$. By Lemma 7, ξ_2 can be transformed into a computation $\xi_2' \in [\![\mathsf{MGC}(L_2)]\!] I_l$ such that $\mathsf{history}(\xi_2') = \mathsf{history}(\xi_1) = \mathsf{history}(\eta)$. We then use Lemma 8 to compose the library-local computation ξ_2' with the client-local one η into a computation $\tau_2 \in [\![C(L_2)]\!] I$ such that $\mathsf{client}(\tau_2) = \mathsf{client}(\eta) = \mathsf{client}(\tau_1)$. □

6 Checking Linearizability on TSO

We have implemented a tool called LINTSO for systematically testing concurrent libraries for our notion of linearizability. Our intention in implementing the tool is twofold. First, the tool allows developers of concurrent libraries to find violations of linearizability quickly. The second (and more important) goal is to use the tool to perform a sanity check of our definition of linearizability by making sure that real-world algorithms that are commonly accepted as correct are linearizable with respect to it.

LINTSO is similar in spirit to the LINE-UP tool for checking linearizability on a sequentially consistent memory model [4]. It takes as input a concrete and an abstract implementation of a library (such as the ones in Figures 3 and 4) along with a (bounded) test harness that calls into the library. LINTSO then composes the input with an operational model of TSO such that sequentially consistent behaviors of the resulting program emulate TSO behaviors of the input. This allows LINTSO to use existing model checkers, such as CHESS [13], to systematically enumerate the behaviors of the harness and the library on TSO.

In a first phase, LINTSO exhaustively generates all histories of the input harness calling into the abstract version of the library. In a subsequent phase, LINTSO systematically enumerates the TSO behaviors of the harness and the concrete version of the library. For every such behavior, LINTSO uses the linearizability condition to check if the behavior is consistent with respect to some history observed in the first phase. Any violation is reported as an error.

If the enumeration in the second phase completes, then LINTSO guarantees that the abstract implementation linearizes the concrete one for the given harness. If the number of possible computations in this phase is too large, a subset of them can be considered by bounding the number of context switches [13]. Obviously, this does not provide a complete guarantee of linearizability, as only (possibly a subset of computations of) one of the infinitely many harnesses is considered.

In our experiments we considered the following concurrent algorithms that were identified as challenges in [14]:

- seqlock, the readers-writer lock we discussed in Section 4;
- simple spinlock, which does not provide fairness guarantees;
- ticketed spinlock, ensuring fairness using a variant of the Bakery algorithm;
- initialisation using double-checked locking.

We provide their code and specifications in [5, Appendix B]. The seqlock and the spinlock implementations are used in various versions of the Linux kernel [3]. The above algorithms are optimised for the TSO memory model and, when used in certain ways, can exhibit behaviours that cannot be reproduced on a sequentially consistent memory model. In fact, the correctness of the spinlock implementations was a subject of debate among Linux developers [14].

In more detail, the simple and ticketed spinlocks do not execute a memory barrier after writing a value into the lock data structure saying that the lock is free. According to the semantics of TSO, this does not violate mutual exclusion: delaying the write in the store buffer can only lead to CPUs that want to acquire the lock waiting longer. As in the case of a seqlock (Figure 4), the specification of a spinlock captures the fact that the lock release can be delayed.

The initialisation using double-checked locking first checks if an object is initialised by reading a corresponding flag without acquiring the lock for the object. Since the read is not preceded by a memory barrier, on TSO this can cause it to return 'uninitialised' even after the object has been in fact initialised. This does not violate the correctness of the algorithm, since the flag is then re-checked with the lock held.

For simple harnesses of the above examples, consisting of up to 3 threads, each performing up to 3 operations, LINTSO performed the check in a matter of minutes. The specification histories were generated exhaustively, and the implementation histories for computations up to a maximum of two preemptions (the CHESS default). In all cases, the tool did not detect any errors. As a further sanity check, we introduced simple errors in the examples, e.g., by replacing xunlock with an unlock in the concrete version. LINTSO was able to find all of them.

We used Theorem 4 to modularise checking the linearizability of the intialisation using double-checked locking. Namely, Theorem 4 allowed us to consider the specification of the spinlock used in this example, instead of a particular implementation. This cut down the number of interleavings to be analysed and made the analysis more efficient. Additionally, it allowed us to prove the linearizability of the algorithm regardless of the particular spinlock implementation used (e.g., the simple or ticketed spinlock). This is just one example of using the Abstraction Theorem to verify concurrent programs compositionally.

7 Related Work and Conclusion

All the definitions of linearizability proposed for various settings so far [11,8,10,9] have assumed a sequentially consistent memory model. This paper is the first to define a notion of linearizability on a weak memory model and show that it validates the Abstraction Theorem (Theorem 4). Our result is based on a novel insight about what information should be kept in histories to specify interactions between the library and the client due to the weak memory model. Even though in this paper we considered only one weak memory model—TSO, implemented by x86 processors [15]—our insights form a starting point for investigating weaker memory models, such as those of Power [17] and ARM [1] processors, and the C++ language [2].

Our work lays the foundation for future correctness proofs for implementations of concurrent algorithms in operating system kernels [3] and language run-times [2]. In particular, we hope that it should be possible to develop a logic for establishing the proposed notion of linearizability formally, based on existing logics for proving safety properties on TSO [20,16] and linearizability on sequentially consistent memory models [18,19]. This should make proofs such as that of Theorem 3 easier to carry out.

We also intend to investigate definitions of linearizability on weak memory models in cases when the library and the client interact in more complicated ways. For example, in this paper we did not consider the transfer of data structure ownership between the library and the client, assuming that they communicate only by passing values of a primitive type. We believe that our approach to handling weak memory can be married with a previous generalisation of linearizability for ownership transfer on a sequentially consistent memory model [9].

Acknowledgements. We thank Scott Owens, Ian Wehrman and the anonymous reviewers for comments that helped to improve the paper. Yang was supported by EPSRC.

References

1. Alglave, J., Fox, A., Ishtiaq, S., Myreen, M.O., Sarkar, S., Sewell, P., Zappa Nardelli, F.: The semantics of Power and ARM multiprocessor machine code. In: DAMP (2009)
2. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: POPL (2011)
3. Bovet, D., Cesati, M.: Understanding the Linux Kernel, 3rd edn. O'Reilly (2005)
4. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: A complete and automatic linearizability checker. In: PLDI (2010)
5. Burckhardt, S., Gotsman, A., Musuvathi, M., Yang, H.: Concurrent library correctness on the TSO memory model, extended version (2012), `http://www.software.imdea.org/~gotsman`
6. Burckhardt, S., Musuvathi, M.: Effective Program Verification for Relaxed Memory Models. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 107–120. Springer, Heidelberg (2008)
7. Cohen, E., Schirmer, B.: From Total Store Order to Sequential Consistency: A Practical Reduction Theorem. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 403–418. Springer, Heidelberg (2010)
8. Filipović, I., O'Hearn, P., Rinetzky, N., Yang, H.: Abstraction for Concurrent Objects. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 252–266. Springer, Heidelberg (2009)
9. Gotsman, A., Yang, H.: Linearizability with ownership transfer. Draft (2011), `http://www.software.imdea.org/~gotsman`
10. Gotsman, A., Yang, H.: Liveness-Preserving Atomicity Abstraction. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 453–465. Springer, Heidelberg (2011)
11. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. TOPLAS 12 (1990)
12. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL (2005)
13. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: OSDI (2008)
14. Owens, S.: Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 478–503. Springer, Heidelberg (2010)

15. Owens, S., Sarkar, S., Sewell, P.: A Better x86 Memory Model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009)
16. Ridge, T.: A Rely-Guarantee Proof System for x86-TSO. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 55–70. Springer, Heidelberg (2010)
17. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: PLDI (2011)
18. Vafeiadis, V.: Modular fine-grained concurrency verification. PhD Thesis. Technical Report UCAM-CL-TR-726, University of Cambridge (2008)
19. Vafeiadis, V.: Automatically Proving Linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010)
20. Wehrman, I., Berdine, J.: A proposal for weak-memory local reasoning. In: LOLA (2011)